MEDICAL DEVICE SECURITY THROUGH HARDWARE SIGNATURES

A Dissertation
Presented to
The Academic Faculty

By

Taimour Wehbe

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

December 2018

MEDICAL DEVICE SECURITY THROUGH HARDWARE SIGNATURES

Approved by:

Dr. Mooney, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Keezer, Co-Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Inan
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Saltaformaggio
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Pande
School of Computer Science
Georgia Institute of Technology

Dr. Iskander
Advanced Security Research Group
Cisco Systems

Date Approved: October 25, 2018

"You can't defend. You can't prevent. The only thing you can do is detect and respond."

– Bruce Schneier

To my beloved parents,

Ziad and Souhaila

and

to my special and amazing wife,

Jihan

to whom I am forever indebted

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my advisor, Prof. Vincent J. Mooney III, for without his guidance and persistent help, this dissertation would have not been possible. I would also like to thank my coadvisor, Prof. David C. Keezer, for his encouragement, guidance and support especially in the early stages of my Ph.D. studies. Thank you both for your forthcoming novel ideas and suggestions and for providing me with your support and patience throughout my research.

Besides my advisor and co-advisor, I would also like to thank my Ph.D. dissertation committee members, Prof. Omer T. Inan, Prof. Brendan Saltaformaggio, Prof. Santosh Pande and Dr. Yousef Iskander, for their valuable suggestions and insightful comments.

I am also grateful for the Graduate Teaching Assistantship positions provided by the School of Electrical and Computer Engineering at the Georgia Institute of Technology. Without this source of financial support, I would have not been able to complete my research work and pursue my Ph.D. degree. I would also like to extend my thanks to the Cisco University Research Program Fund as part of the Silicon Valley Foundation for their generous gift donation which provided my final one and a half years of financial support towards my Ph.D. degree.

I have also been very fortunate to meet some of the smartest engineers and engineers-to-be at Georgia Tech. I would like to thank all the graduate and undergraduate students who helped in any form with my Ph.D. work. Special thanks go to Yanshen Su, Chinmoy Kulkarni and Ethan Lyons for helping in the implementation of several of the techniques developed in my work. I would also like to thank Prof. Inan and his students, especially Abdul Qadir Javaid, for providing me with captured health data of multiple anonymous individuals to help test the developed architectures in this work using real-life data.

Last but not least, I could not express how grateful I am to my parents, Ziad and Souhaila, to whom I am forever indebted, and my amazing wife, Jihan, for keeping up

with me during the long days and nights. Finally, I would like to thank my family, my friends and all those who contributed in any shape or form to the success of this work.

It is because of all of these amazing people that I am here today. To all of you, I say, thank you!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

3D Three Dimensional

A/D Analog-to-Digital

ADC Analog-to-Digital Converter

ALU Arithmetic Logic Unit

ARM® Advanced RISC Machine

ASIC Application Specific Integrated Circuit

ASLR Address Space Layout Randomization

AXI Advanced Extensible Interface

BCG Ballistocardiogram

BILBO Built-In Logic Block Observer

BRAM Block RAM

CBC Cipher Block Chaining

CPI Cycles Per Instruction

CT Ciphertext

CUT Circuit Under Test

CVE Common Vulnerabilities and Exposures

CWE Common Weakness Enumeration

DDR Double Data Rate

DLL Dynamically Linked Library

DMA Direct Memory Access

DRAM Dynamic RAM

DRC Design Rule Check

ECG Electrocardiogram

ELF Executable and Linkable Format

ERC Electrical Rule Check

FDA Food and Drug Administration

FIFO First-In-First-Out

FPGA Field Programmable Gate Array

GE Gate Equivalent

GOT Global Offset Table

GP General Purpose

HF Head-to-Foot

HMAC Hash-based MAC

HP High Performance

HT Hardware Trojan

HW Hardware

IC Integrated Circuit

IEEE Institute of Electrical and Electronics Engineers

IMU Integrity Measurement Unit

IO Input Output

IoT Internet-of-Things

IRB Institutional Review Board

JIT Just-In-Time

KASLR Kernel ASLR

LFSR Linear Feedback Shift Register

LSB Least Significant Bit

LUT Look Up Table

LVS Layout Versus Schematic

MAC Message Authentication Code

MISR Multiple Input Signature Register

MSB Most Significant Bit

ORA Output Response Analyzer

OS Operating System

PFN Page Frame Number

PL Programmable Logic

PLT Procedure Linkage Table

PRPG Pseudo-Random Pattern Generator

PS Processing System

PT Plaintext

PUF Physically Unclonable Functions

RAM Random Access Memory

RISC Reduced Instruction Set Computer

ROP Return-Oriented Programming

SHA Secure Hash Algorithm

SGX Software Guard Extensions

SNR Signal-to-Noise Ratio

SoC System-on-Chip

SPI Serial Peripheral Interface

TOCTOU Time-of-Check to Time-of-Use

TPG Test Pattern Generator

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

XOR Exclusive-OR

SUMMARY

Software and hardware attacks on embedded and medical devices can cause serious harm if not quickly detected. This dissertation presents techniques based on hardware signatures aiming to address the part of the attack surface which entails inserting malicious hardware circuitry (Hardware Trojans) during the manufacturing process of a digital microchip and maliciously modifying executable code at run-time.

On the hardware side, the type of Hardware Trojan (HT) discussed in this work is composed of a few gates and attempts to modify the functionality of the chip. Such types of extremely small HTs are hard to detect using other conventional offline HT detection methods, such as side-channel analysis and digital systems test techniques. Our novel approach, however, focuses on an online method for rapidly detecting HTs at run-time by checking for the correct functionality of the underlying hardware. We present an architecture that addresses these threats by splitting the design into a two-chip approach where we generate signatures in the hardware at the very beginning of data harvesting, and we then check for these signatures during data processing and encryption. In addition, we take advantage of known physiological relationships between medical data to ensure the integrity of the data that is processed by the hardware.

On the software side, techniques that detect attacks on application code at run-time typically rely on software due to the ease of implementation and integration. However, these techniques are still vulnerable to the same attacks due to their software nature. In this work, we present a novel hardware-assisted run-time code integrity checking technique where we aim to detect if executable code resident in memory is modified at run-time by an adversary. Specifically, a hardware monitor is designed and attached to the device's main memory system. The monitor creates page-based signatures (hashes) of the code running on the system at compile-time and stores them in a secure database. It then checks for the integrity of the code pages at run-time by regenerating the page-based hashes (with

unmapped regions zeroed out) and comparing them to the legitimate hashes. The goal is for any modification to the binary of a user-level or kernel-level process that is resident in memory to cause a comparison failure and lead to a kernel interrupt which allows the affected application to halt safely. We are able to check the majority of executable code with the exception of a few page table entries to redirect application code to libraries.

Our experimental results demonstrate the efficiency and effectiveness of our proposed and implemented techniques. Specifically, our HT detection architecture was able to not only detect HT attacks but also distinguish these attacks from actual health problems. In addition, our run-time code integrity checking technique was able to rapidly detect zero-day malware attacks while introducing minimal resource overhead and negligible performance degradation on applications running on an embedded device such as a heart rate monitoring application.

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

The rapid technological advancement that we are featuring in our lives has led to an increasing reliance on technology in our everyday activities. With the emergence of the Internet-of-Things (IoT), we are seeing embedded devices become increasingly interconnected and widespread spanning the range of applications from simple entertainment consumer electronics to complex and safety critical applications such as medical devices, driverless cars and smart power grids. Attacks on such highly-connected embedded devices have increased significantly in the past decade. The majority of these attacks have come from the fact that embedded devices are typically resource-constrained and cannot afford the luxury of having full-blown security primitives implemented on them. Moreover, the connection of these resource-constrained devices to the public internet opens an extensively increasing plethora of software vulnerabilities including run-time code modifying attacks that attempt to maliciously change the behavior of running applications on an embedded device's operating system (OS).

In addition, embedded devices have been targets of malicious insertion of undesirable logic functions into the hardware of fabricated digital chips. Such malicious behavior has been facilitated by the disaggregation of the chip manufacturing industry. The aforementioned stealthy hardware modifications are referred to in the literature as Hardware Trojans (HTs). HT attacks have been shown to have the ability to leak sensitive information or even alter the functionality of a system. HTs could be inserted during any stage of the fabrication process. For example, Figure 1.1 shows that even if the design team is trusted, an insider can insert an HT during physical design and verification or even during fabrication. The

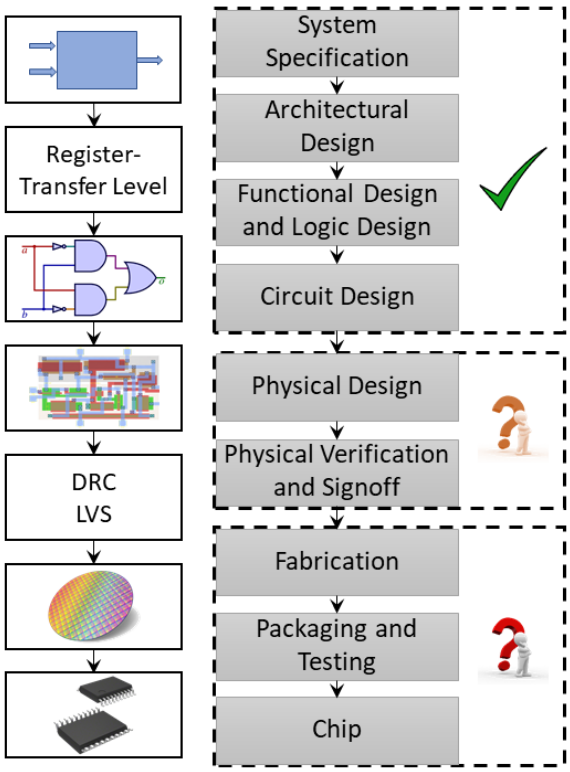


Figure 1.1: The disaggregation of the chip manufacturing industry has led to increasing possibilities of malicious hardware modifications.

effects of HTs can be disastrous if the attack targets sensitive applications such as medical or embedded military devices. For example, in 2010, the United States Navy discovered missiles provisioned with fake microchips with a "back door" that could have been used to remotely shut the missiles down at any time, rendering them useless [1]. Moreover, a very recent report released by Bloomberg in early October 2018 claims that the supply chain of a major server motherboards manufacturer has been compromised by the insertion of a back-door chip [2]. Over the past decade, research agencies and groups have been looking for ways to verify that digital chips are not hacked during the production process. For instance, in 2016, the Food and Drug Administration (FDA) and the IEEE Center for Secure Design released reports spotlighting security red flags for the wearable industry, one of which is falsifying a user's health data by physically manipulating the device [3, 4].

Designing techniques to detect hardware and software attacks in such small embedded

devices has proven to be a non-trivial task. On the hardware side, techniques that try to perform offline testing, such as side-channel analysis or digital systems test, have so far not been able to guarantee the security of the chip. In addition, online techniques that try to detect HT attacks at run-time need careful design considerations due to the energy consumption and computing power limitations for these devices.

On the software side, while conventional security measures, such as security protocols/standards and cryptographic algorithms, provide a strong basis for securing embedded systems by addressing security considerations from a theoretical perspective [5], relying solely on conventional approaches has proven to be ineffective as newer trends have shown that most attacks take advantage of weaknesses present in the implementation of an embedded device. Specifically, a system's security can be compromised through corruption of binaries as they are being downloaded or stored on the embedded system or through the execution of untrusted or unknown sources. Techniques that perform checking of executable code at compile and load time have been widely spread and proven to be effective [6–8]. However, techniques that verify correct run-time execution are still a major challenge, especially when targeting resource-constrained embedded devices [9–12]. For example, consider an operating system (OS) that is running a user-level application. The system typically starts by loading the application code from disk to memory. Although there have been a plethora of techniques that ensure the integrity of the code while it is present on disk and right before load-time, verifying the correct execution of that same code while it is resident in memory presents new challenges. Malicious activities, such as malware running on a system, can try to modify the code content at run-time. In fact, these types of attacks have recently become more prevalent. For instance, G. Holmes classified malware in compromised devices into five variants [13], three of which use a run-time infection method to modify and/or insert malicious code where modifications are made to the in-memory copy of the executable code.

1.2 Sample Embedded Medical Application

Electrocardiogram (ECG) and Ballistocardiogram (BCG) sensors are normally used to monitor and capture heart pulse electrical and timing events. Figure 1.2 presents a health monitoring scenario showing a person standing on a BCG scale and holding on to an ECG handlebar. The BCG force-plate adjusts itself with pumps to non-invasively capture three-dimensional BCG forces that represent the cardiogenic vibrations of the body [14–16]. The ECG handlebar is used as a grip-style dry electrode for ECG measurement [17]. The person's cellphone (shown in the figure) is used to capture 3D on-body inertial measures to help adjust the person's postural sway and generate improved and more accurate ECG and BCG readings. Therefore, the BCG hardware is running a real-time application with data processing to provide the necessary balancing mechanisms. In this work, the ECG signal along with the BCG signals are used to create the necessary hardware signatures that check for the integrity of the captured data. Harvested ECG and BCG data are typically further processed and analyzed to extract physiological features that assist in the diagnosis of health conditions [14, 15, 18].

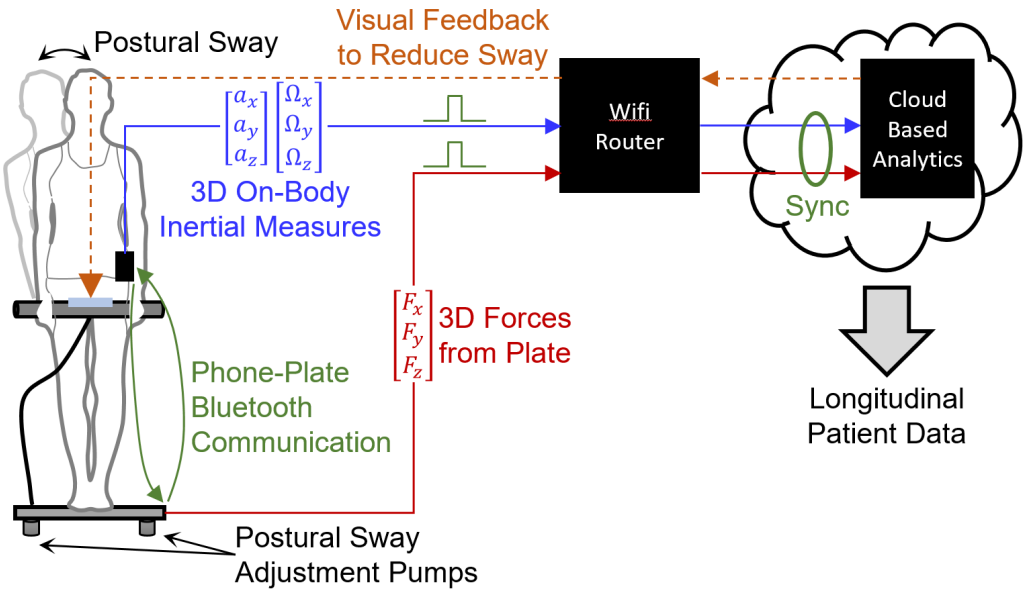


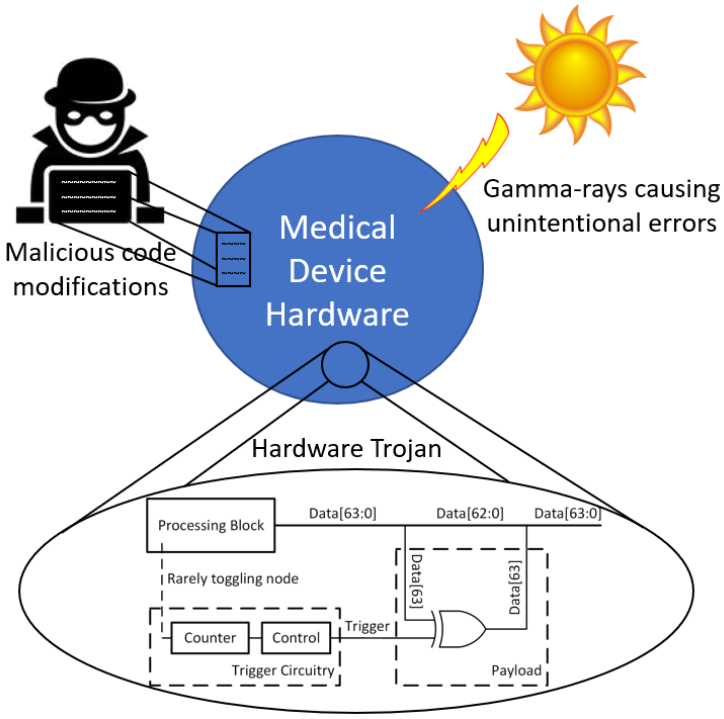
Figure 1.2: A medical device example showing a patient's heart signals being captured in real time.

The presented application shows the importance of providing security against malicious entities not only at the software level but also at the hardware level. For example, an attack on the ECG or BCG data as it is being harvested could result in maladjustment of the postural sway pumps of the BCG scale potentially affecting a patient's balance and causing harm. Therefore, it is imperative to present security solutions to such resource-constrained embedded devices to not only protect against the integrity of the data as it is being captured but to also protect the software performing the computation at run-time.

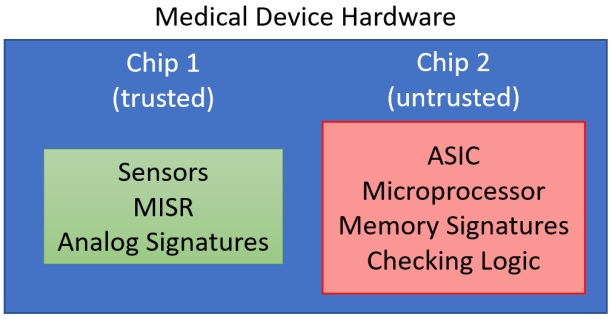
1.3 Research Overview

Figure 1.3a presents a high level overview of the attacks and errors that are targeted in this research. Specifically, the research addresses HT attacks, hardware errors and malicious software code modifications in embedded medical devices. Figure 1.3b shows an overview of our novel architecture where medical device hardware is partitioned into two chips. The first chip (Chip 1) is responsible for capturing sensor data and performing initial signature generation, while the second chip (Chip 2) is responsible for checking for the integrity of the captured and processed data. The presented architecture targets HTs that are extremely small in size and which, once triggered, attempt to modify the functionality of the chip by attacking the user's data. This work is motivated by the health monitoring scenario presented in Section 1.2 which includes sensors capturing heart signals and transmitting them for further processing and analysis [14]. The physiological signals have known relationships which are used to create multiple types of signatures that assess the integrity of the captured data at run-time [19–22].

In addition, this work presents a novel hardware-assisted run-time code integrity checking technique which ensures that code resident in memory is protected at run-time from any malicious modification [23]. Specifically, we perform run-time memory monitoring through a separate and isolated hardware monitor. Page-based signatures (memory signatures), which are stored in a system's root-of-trust, are generated at compile-time and are



(a) Targeted attacks and errors



(b) Designed and implemented architecture

Figure 1.3: An overview of the targeted attacks and errors on embedded medical devices and the designed and implemented architecture to address these attacks.

then checked at run-time using the dedicated hardware monitor. Any modification to the binary of a process that is resident in memory will cause a comparison failure and lead to a kernel interrupt which will eventually cause the affected application to halt safely.

Our work improves consumer confidence in health monitoring applications by increasing the assurance provided to the public that their captured health data are correct. For example, consider a scenario where health data collected by such applications are used

to interact with drug titration procedures (i.e., where medical dosage is varied dynamically and without necessarily visiting a doctor). An HT attack or a malicious software in-memory code modification can cause misleading analysis of the patient's health leading to possible lawsuits. Providing confidence in data readings reduces putting a person's health in danger and may also provide legal protections, i.e., additional legal defense.

1.4 Dissertation Organization

The remaining chapters of this dissertation are organized as follows.

Chapter 2 presents some background and prior work related to hardware Trojan research and code integrity research. An overview of the different types of hardware and software attacks is presented along with a survey of the current state-of-the-art techniques present in the literature to address such types of attacks.

Chapter 3 discusses in detail the specific hardware and software threat models that are addressed in this dissertation. The assumed triggering methods, the payloads and the effects of such attacks on the hardware and software of an embedded medical device are presented.

Chapter 4 introduces one of the two major contributions of this dissertation. Specifically, a novel HT detection technique based on the use of hardware signatures is presented to assess the integrity of a patient's health data as it is being harvested by medical sensors. Details of the sub-components of the designed and implemented method are then explained followed by an analysis of how the architecture is set up to detect specific types of the HT attacks presented in our threat model.

Chapter 5 presents the second major contribution of this dissertation. The chapter starts by introducing a methodology for assessing run-time code integrity and detecting malicious modifications to code performing computation on user's data. A hardware-assisted architectural implementation of the presented methodology is then described in detail along with the challenges of implementing such an architecture on an embedded system.

Chapter 6 presents the experimental setup, hardware implementation, results and anal-

ysis showing the effectiveness of the two presented architectures on assessing the security and integrity of both data and computation in embedded medical devices.

Chapter 7 highlights the advantages of the presented methodologies in this dissertation as it pertains to the impact of these techniques on embedded systems security in general. Specifically, some example case scenarios are discussed showing how our hardware signature-based architectures can be used to implement chip-level security frameworks for assessing data integrity in generic sensor nodes. Moreover, the chapter discusses some open research questions and avenues for improving the presented techniques in this dissertation.

Finally, Chapter 8 concludes the work presented in this dissertation and highlights the major contributions of this research.

CHAPTER 2
BACKGROUND AND PRIOR WORK

2.1 Introduction

In this chapter, we present some background and literature review regarding the two major topics of this dissertation. We first start by introducing Hardware Trojan research by providing a summary of the known types of attacks along with a survey of the detection techniques presented in the literature. We then provide background related to run-time code integrity and present a survey of the different types of code integrity attacks along with the software, hardware and hardware/software codesign detection mechanisms. In addition, the advantages and disadvantages of each of the hardware and software detection techniques are highlighted. Finally, before concluding this chapter, we present some background about hardware signature generation and digital systems test as they relate to the hardware primitives utilized in the techniques presented in this dissertation.

2.2 Hardware Trojan Research

Hardware Trojan attacks on highly interconnected embedded devices can cause severe damage, especially in the medical field. Techniques to detect and try to prevent these HTs from infiltrating digital chips have been receiving increased attention in the past several years.

2.2.1 Classification and Attacks

Over the past decade, a more formal HT taxonomy has been introduced [24]. Figure 2.1 shows how different types of HT attacks can be classified according to three broad characteristics: (i) physical characteristics, (ii) activation characteristics and (iii) action characteristics [24]. The (i) physical characteristics divide the HT attacks according to their dis-

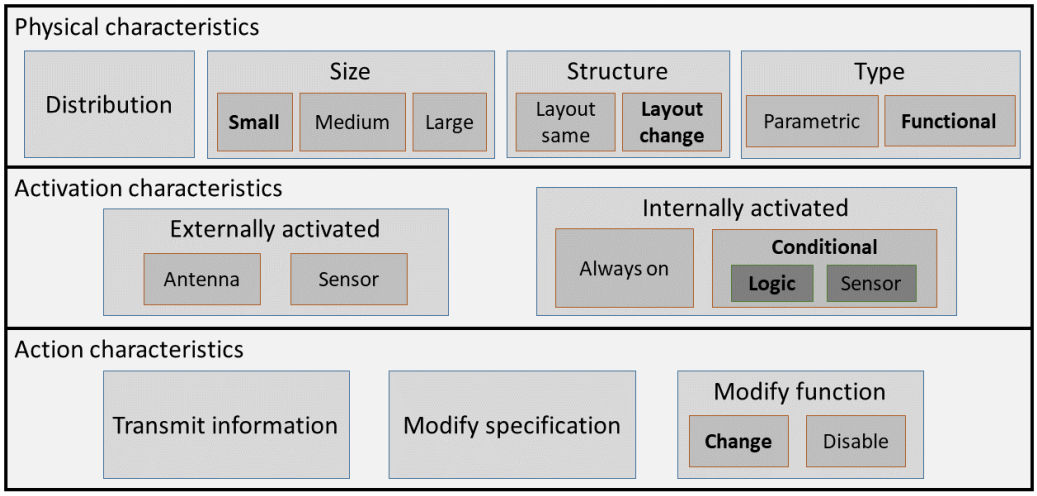


Figure 2.1: A Hardware Trojan taxonomy characterizing HTs according to their physical, activation and action properties.

tribution on the chip, their size (small, medium, or large), their structure (layout changes) and their type (parametric or functional). The (ii) activation characteristics divide them into internally activated HTs and externally activated ones. Externally triggered HTs wait for an activation signal coming from outside the chip. Internally triggered ones can be classified into two subtypes: (a) an always on HT and (b) a conditionally triggered HT with the latter having the condition dependent on some logic in the circuit or some sensor attached to the circuit. The (iii) action characteristics divide the HT attacks according to their effect, i.e., whether the HT is going to leak information, modify the circuit's specification, and/or corrupt or disable functionality.

2.2.2 Detection Mechanisms

Designing techniques to efficiently detect malicious hardware modifications in embedded devices has proven to be a non-trivial task. Techniques that try to perform offline testing for HTs, such as side-channel analysis or digital systems test, have so far not been able to guarantee the security of digital chips. In addition, online techniques that try to detect HT attacks at run-time need careful design considerations due to the energy consumption and computing power limitations for these devices.

Quite a few HT detection methods [19–22, 24–34] have been proposed to address specific types of the aforementioned attacks. These methods can be broadly divided into side-channel analysis techniques, HT triggering techniques and correct functional verification techniques.

2.2.2.1 Side-channel Analysis

This type of HT detection relies on analyzing information gained from the physical implementation of an architecture. The information gained from side-channels are typically compared to data generated by the normal behavior of a chip to detect anomalies. Energy consumption, timing analysis and electromagnetic emanations are the most common techniques for side-channel analysis. The authors of [24–27] provide multiple types of HT detection methods including methods that are based on power analysis and timing analysis. These types of methods lack the ability to detects HTs that are small in size such as the ones introduced in [28, 29] due to the minor effects that the HT might cause in terms of power and timing variations.

2.2.2.2 Hardware Trojan Triggering

This type of HT detection relies on extensively testing the design prior to deployment and use (i.e., right after chip fabrication). Prior work has proposed methods to detect HTs with significant hardware footprints, e.g., using a "golden die." For example, in [30], the authors introduce an HT prevention and detection mechanism for integrated circuits (IC) where they prevent a wide variety of HT attacks during IC testing and system operation in the field. However, the presented mechanism relies on comprehensive schemes of special error detecting codes resulting in increased hardware overhead of up to 165% [30]. Also, the authors of [31, 32] provide a survey of multiple types of HT detection methods including methods that are based on HT activation mechanisms. HT detection approaches based on a "golden die" present a variety of disadvantages. First, cleverly inserted HTs may

not be easily triggered by these approaches as the testing mechanisms have no idea of the presence of the HT and/or its location in the chip. Thus, the HT might pass the testing phase undetected. Second, this type of detection does not guarantee the correctness of the design at run-time. Finally, performing an offline full-functionality test for each fabricated chip might turn out to be inefficient and time-consuming, thus increasing the cost of microchips.

2.2.2.3 Functional Verification

This type of HT detection relies on checking the functionality of the hardware by monitoring the output and checking for expected behavior. The authors of [24, 31] provide a survey of multiple types of HT detection methods including methods that are based on architecture-level detection and functional verification. Others have devised techniques that solely rely on functional verification for checking the trustworthiness of the underlying hardware [33, 34]. The work presented in this research falls under this category where hardware signatures are created to check for the correct functional behavior of the digital microchip at run-time [19–22]. However, while the techniques presented in prior research are able to detect HT attacks on internal components of a digital microchip, it is important to note that, to our knowledge, this work is the first to address an HT threat where input values are immediately altered as they initially appear on a digital microchip.

2.3 Code Integrity Research

Checking the integrity of application code running on an operating system (e.g. Linux) has been a long researched topic. Several methods have been proposed including software-only methods and hardware-assisted ones. In addition, these techniques are broadly divided into ones that look for malicious modifications in executable code prior to load-time and ones that provide run-time code monitoring.

2.3.1 Software Attacks

A major portion of attacks on embedded devices is due to the injection of malware especially with the increasing internet connectivity of these devices. Malware attacking such devices can be divided into several categories such as viruses, Trojan horses, spyware, rootkits and other intrusive code [7]. Each of these types of malware performs a specific goal whether it is affecting an application's behavior, leaking sensitive information, spreading network traffic to cause denial of service, or spying on some user's activity.

To better understand and study the weaknesses that these attacks exploit, a division of these vulnerabilities into major classes exists in the Common Vulnerabilities and Exposures (CVE) and Common Weakness Enumeration (CWE) standards developed by the MITRE Corporation [35, 36]. Some of the major classes involved with embedded systems security are buffer errors, code injection, information leakage, permissions, privileges, access control and resource management. Buffer error vulnerabilities are mainly introduced by allowing code to directly access memory locations outside the bounds of a memory buffer that is being referenced. Code injection weaknesses are usually exploited due to the lack of verification of what constitutes data and control for user-controlled input leading to the injection of inappropriate code changing the course of execution. Information leakage weaknesses are introduced when the system intentionally or unintentionally discloses information to an actor that is not explicitly authorized to have access to that information through sent data or through data queries. Permissions, privileges, and access control are weaknesses introduced due to improper assignment and enforcement of user or resource permissions, privileges, ownership and access control requirements. Finally, resource management weaknesses are mainly related to improper management and use of system resources, such as making resources available to untrusted parties and improperly releasing or shutting down resources.

2.3.2 Software and Hardware Countermeasures

A wide variety of techniques have been proposed in the literature to detect malicious modifications to processes running on embedded systems. These techniques are broadly divided into anomaly-based detection and signature-based detection methods where the former is involved in detecting abnormal behavior of operation after the detection engine has learned what forms a safe environment, while the latter is involved in looking for known patterns in which an adversary performs the attack on the system [7, 37]. Both of these techniques have their own advantages and disadvantages. On the one hand, while the anomaly-based detection techniques help in capturing new (zero-day) attacks, they usually introduce undesirable false positives since a simple deviation from expected behavior could potentially trigger an alarm. On the other hand, signature-based detection techniques provide a good approach to capturing known attacks; however, they are unlikely to detect a new attack if the attack's signature is not present in the signature database. However, a majority of these techniques have been mostly or fully implemented in software and are thus vulnerable to software attacks.

2.3.2.1 Compile-time and Load-time Code Inspection

Traditionally, code integrity in computer systems and embedded devices has been provided by a Linux Integrity Measurement Unit (IMU) [6]. IMUs typically verify the integrity of executable content in an operating system at load-time by inspecting the integrity of executable files before loading them.

Plenty of research has been done in this area alongside others such as static analysis software-based techniques that try to find possible security vulnerabilities in the code. However, all these techniques are not able to detect or prevent run-time attacks [6, 10]. A survey of common code injection vulnerabilities and software-based countermeasures is presented in [38]. One common weakness among these types of code inspection methods is the infeasibility of discovering all vulnerabilities in a given program by automated static

analysis alone [39–42].

2.3.2.2 Run-time Code Inspection

Techniques that attempt to detect malicious modifications to application code at run-time have been highly relying on software due to the ease of implementation and integration. However, these techniques are still vulnerable to the same attacks due to their software nature. For example, extensions to the IMU have been added to implement dynamic integrity measurement and to detect and prevent return-oriented programming (ROP) attacks [9, 10, 43]. However, such types of implementations use some form of tracking code and therefore are implemented purely in software which keeps the tracking code vulnerable to attacks. In addition, the IMU extensions' support for run-time detection comes at the expense of performance overhead.

Dynamic software-based techniques usually augment the code by adding some run-time checks so that an attack can be detected. These techniques require either the modification of the target code by adding a new number of executed instructions or the implementation of a separate software monitor in the form of a protected process to keep track of the propagation of data and control during program execution. Therefore, these techniques either require code recompilation or new monitoring code introduction in which both would eventually incur a significant performance overhead [44–46].

Intel SGX and ARM TrustZone have been developed to secure and protect code integrity by isolating user code and allocating private regions of memory [47, 48]. These techniques are complimentary to our presented technique; however, they serve a slightly different goal. Their focus is to separate running applications into secure (trusted) and non-secure (non-trusted) worlds, thus preventing potential attacks on applications running in the secure world, while our work focuses on providing a mechanism to rapidly detect attacks on executable code of any running application. In addition, the implementation of such techniques on resource-constrained devices might turn out to be a challenging task.

Other research work has introduced a combination of hardware/software and hardware-assisted architectures to monitor programs and look for malicious behavior. In these presented approaches, the methods to detect anomalies depend on monitoring the control flow execution of an application [11, 12, 49, 50] or rely on instruction-based monitoring [51–53]. In the former, static analysis of expected program behavior is extracted and then used by hardware monitors to observe the program's execution trace. Thus, such techniques impose limitations on coding styles and are vulnerable to introducing frequent false positives with any simple deviation from normal behavior, primarily due to the fact that the method relies on having an application follow a predefined control flow without allowing for run-time decision making changes. The latter introduces a significant amount of performance overhead. For example, performing integrity checking on basic blocks as described in [51] results in generating a hash for every set of instructions that fall between two consecutive control transfer instructions.

2.4 Hardware Signature Generation and Digital Systems Test

One of the key technologies we rely on in this work is the ability to compute a digital "signature" of a bitstream where the signature is compact (significantly fewer bits than the associated stream of bits) yet has a very low probability of producing the same signature for a different input bitstream. Therefore, if an HT corrupts either the signature or the input stream, the mismatch can be identified with a high level of certainty.

Bitstream signatures can be generated using different techniques and algorithms. Message Authentication Codes (MACs) are traditionally used to compute and generate signatures for use in data transmission. MACs are broadly divided into two types: Cipher Block Chaining MACs (CBC-MACs) and Hash-based MACs (HMACs). CBC-MACs use cipher blocks to create the signature, while HMACs depend on hash functions to provide data integrity and authentication. One commonly used type of HMAC is the Secure Hash Algorithm (SHA). The main advantage of SHA is that it provides signatures with high security

features. However, this comes on the expense of area and increased energy consumption.

Another type of signature generation typically used in digital systems test is a Multiple Input Signature Register (MISR) [54]. In this work, we take advantage of MISRs to create our signatures. The major incentive behind our use of MISRs is their advantage in terms of area and energy consumption over MACs and specifically SHA; in particular, if the MISRs are already in the chip for test purposes, why not reuse them for security purposes?

The following subsections introduce SHA and a specific type of MISR, the BILBO MISR, followed by a brief discussion comparing their area consumption and security features pertaining to signature generation.

2.4.1 Secure Hash Algorithm (SHA)

In cryptography, messages can be signed by a secure hash algorithm (SHA). Several versions of SHA exist with SHA-2 and SHA-3 [55] considered to be acceptably secure. However, the high security introduced by these algorithms comes at the expense of significant (large) layout area and energy consumption.

For example, in the report of the SHA-3 cryptographic hash competition that is published by NIST [56], it has been stated that the area of full implementations of 256-bit SHA-3 ranged between 39k Gate Equivalents (GE) on 130nm CMOS technology [57] and 80kGE on 65 nm CMOS technology [58]. Both of these designs were optimized for maximum throughput/area. A compact (lightweight) implementation of the 256-bit SHA-3 algorithm was around 15kGE on a 90 nm CMOS technology [59].

2.4.2 Built-In Logic Block Observer (BILBO) Multiple Input Signature Register (MISR)

A Built-In Logic Block Observer (BILBO) is one form of Built-in Self-Test (BIST) techniques in digital systems test [54]. A major feature of BILBO is the use of existing flip-flops in the Circuit Under Test (CUT) to create an architecture that includes Test Pattern Generators (TPGs) and Output Response Analyzers (ORAs) [54]. A BILBO can be configured to

operate as a Multiple Input Signature Register (MISR). MISRs are used in digital systems test to compress a large number of test outputs into a small "signature" so that pass/fail comparison logic is greatly reduced and tests are completed more quickly [54].

Figure 2.2 shows four bits of a reconfigurable n-bit BILBO register. The BILBO register operates in four modes: (i) a normal parallel load mode, (ii) a shift register with scan input "Scan In," (iii) a Multiple Input Signature Register (MISR), and (iv) a Pseudo-Random Pattern Generator (PRPG) as controlled by the four combinations of bits B1 and B2 [54].

A MISR is a variation of a Linear Feedback Shift Register (LFSR) where multiple inputs can be input into the LFSR exclusive-or gates prior to each flop-flop as seen in Figures 2.2 and 2.3. Figure 2.3 shows a four-bit BILBO register configured to operate in MISR mode [54]. Traditionally, MISRs implement test compression by use of a predetermined "primitive polynomial" which results in the MISR generating a small signature with a very low probability (approximately $1/2^n$ for an n-bit MISR) of two different bit sequences producing the same signature [54]. The specific calculation shown in Figure 2.3 is for the architecture to take inputs In[3:0] and compress the response according to a specific primitive feedback polynomial – namely, $x^4 + x + 1$ in Figure 2.3 – to form a signature. There are several published tables with lists of primitive polynomials for multiple MISR sizes [60–62]. For example, a list of primitive polynomials of degree n for every $n < 64$ is provided in [62].

The use of a MISR for signature generation incurs low area overhead since the architecture takes advantage of existing sequential logic and registers. In particular, for test purposes, it is common to add a few gates per bit to change a normal register into a BILBO register as described in [63]. Once the BILBO registers are in the design, we can program them as shown in Figure 2.3 to operate in MISR mode during run-time.

In addition, use of a MISR for signature generation is much simpler than hashing algorithms and produces signatures with reasonable image resistance. Specifically, after a

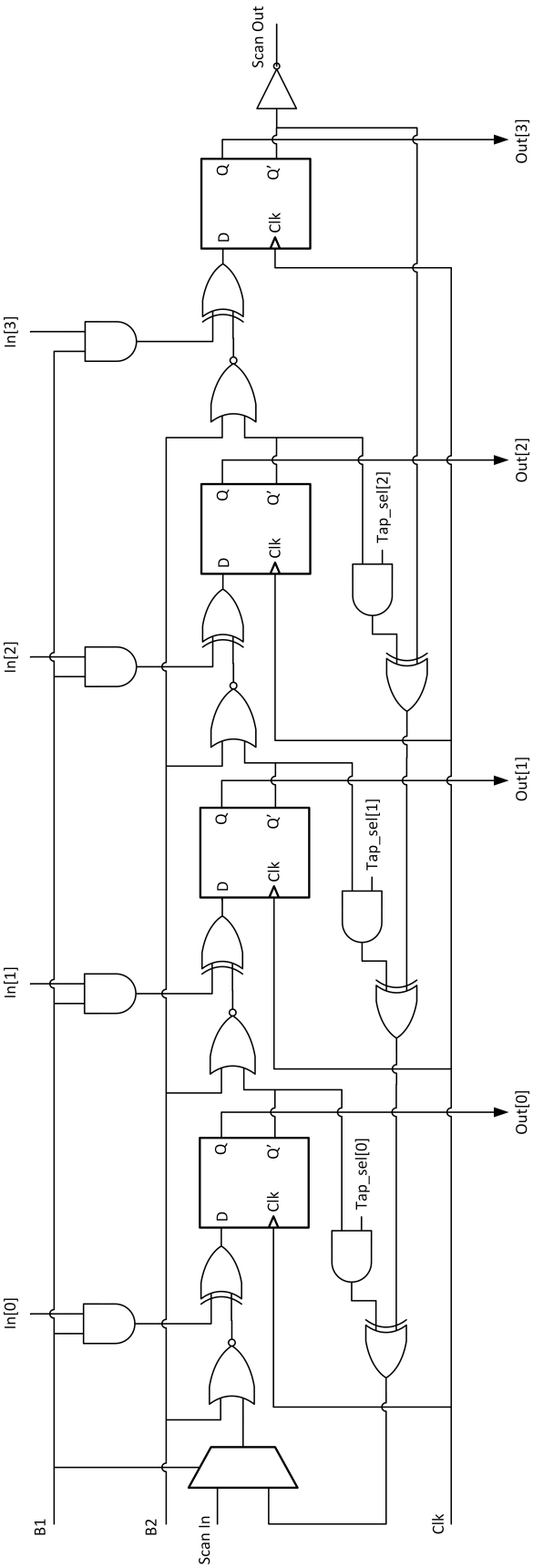


Figure 2.2: A 4-bit reconfigurable Built-In Logic Block Observer (BILBO) register.

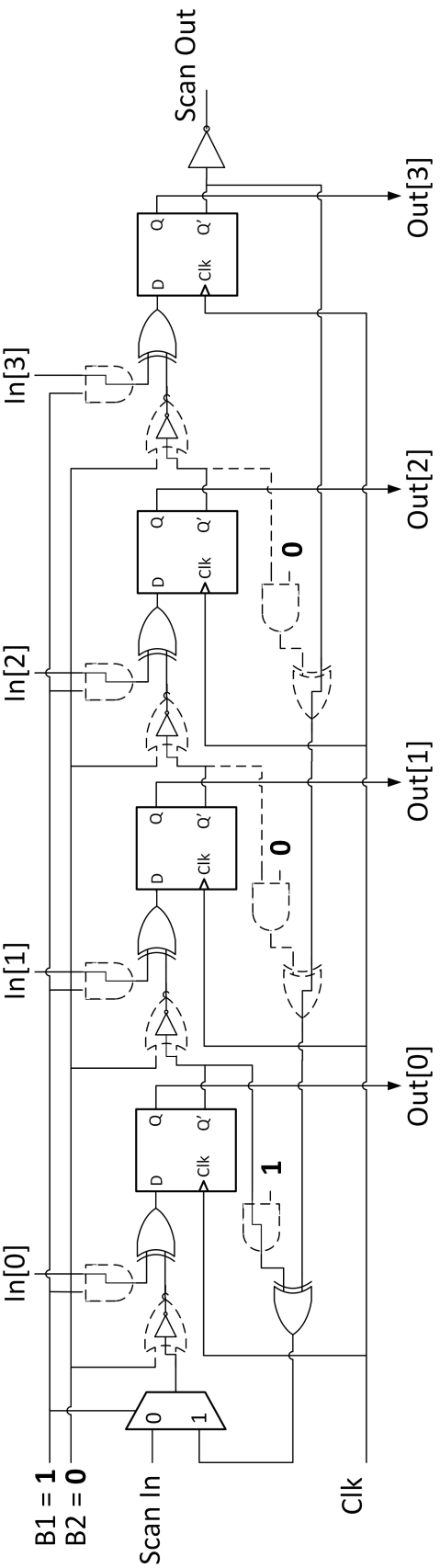


Figure 2.3: A digital systems built-in self test BILBO module configured to operate as a MISR.

sufficient number of clock cycles, the probability of appearance of each pattern becomes $1/2^n$ for an n-bit MISR provided that the number of input patterns to the MISR is greater than one [54]. Thus, for low power applications, the use of MISRs for signatures might be favorable over higher power and area hash algorithms, especially if the MISR's primitive polynomial is carefully chosen.

2.4.3 MISR Encryption

SHA is designed to not give away information about encrypted data to an attacker who obtains the full set of transmitted bits. If we use MISR logic to generate a signature, it is not known whether or not the MISR signature could be used as a side channel to break the encryption. However, some recent work appears to indicate that a suitably modified MISR (e.g., altered to avoid linearity) may have important hash function properties for security [64].

A straightforward way to protect a MISR signature is to encrypt it with the same protocol used to encrypt the messages. In this work, we use PRESENT which is a low-area, low-power block-cipher utilizing either an 80-bit key or a 128-bit key [65]. 128-bit PRESENT has been shown to be as cryptographically secure as 128-bit AES [65].

2.5 Summary

Designing techniques to detect malicious modifications to hardware and software in embedded and medical devices has been a major challenge. Resource, area and power limitations on such devices impose hard constraints on creating these architectures. In this chapter, we present some of the major types of HT detection techniques in the literature along with their corresponding advantages and disadvantages. In addition, we describe some of the most adopted methods and architectures for detecting run-time code integrity attacks in embedded systems and highlight the strengths and drawbacks of such architectures.

CHAPTER 3

HARDWARE AND SOFTWARE THREAT MODELS

3.1 Introduction

This chapter addresses in detail each of the hardware and software threats that are targeted in this dissertation. We first introduce our main Hardware Trojan (HT) threat scenario in which an adversary can confiscate the integrity of a user's data. We then present our software threat model as it pertains to run-time code integrity by stating the types of vulnerabilities and attacks that are addressed by the techniques presented in this work. Finally, we present the threats that are not currently addressed by the work presented in this dissertation.

3.2 Hardware Trojan Threat Model

One of the most interesting classes of HT attacks is the type that stealthily targets the functionality of a specific digital design via minimal logic insertion. This type of attack typically relies on a trigger condition as shown in Figure 2.1. In our work, we consider the threat model shown in Figure 3.1. The HT model, which is representative of most prior work [28–31, 33], is composed of *trigger circuitry* and a *payload*.

As shown in Figure 3.1, the *payload* can be as small as a single exclusive-or (XOR)

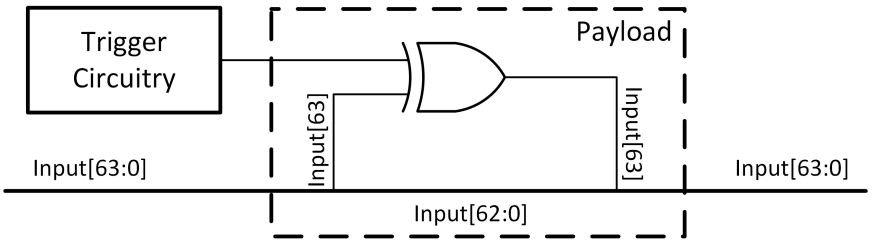


Figure 3.1: Our HT threat model showing trigger circuitry and a payload composed of a simple XOR gate.

gate that modifies data on a bus in the design (*Input* bus signal in this case) by toggling a single bit (most significant bit in this case). As Figure 3.1 shows, such a type of HT attack is extremely small (just a few gates) and cannot be reliably detected if hidden among thousands or even millions of gates in an embedded system design [29]. In addition, the effect of such an attack could be disastrous if it goes undetected for relatively long periods of time. For instance, in the example scenario shown in Figure 1.2 and as discussed in Section 1.2, an attack on the ECG or BCG data could hypothetically cause the adjustment pumps on such a BCG scale to behave abnormally in a way that could harm the person standing on the scale.

Figure 3.2 shows an example of trigger circuitry responsible for waiting for an activation characteristic to trigger the HT. The activation characteristic, as described in Section 2.2.1 and as shown in Figure 2.1, can be externally or internally triggered. In our threat scenario, we consider HT trigger circuitry which is based on a conditionally triggered HT. As Figure 3.2 shows, the trigger circuitry can be designed using only a *counter* and minimal *control* logic. The *counter* is attached to a *rarely toggling node* in the *processing block* on the chip. The *counter* is incremented every time the value on the wire toggles. The *control* logic monitors the output of the counter and asserts the *trigger* once the *counter* reaches a specific predefined value.

It is important to note that in our threat model, we assume that once the HT is triggered, it remains on for a relatively long time. That is because if an attacker wants to intelligently turn on the HT for finite periods to bypass specific detection techniques, the trigger circuitry (shown in Figure 3.2) will have to be more complex resulting in an HT with a larger size.

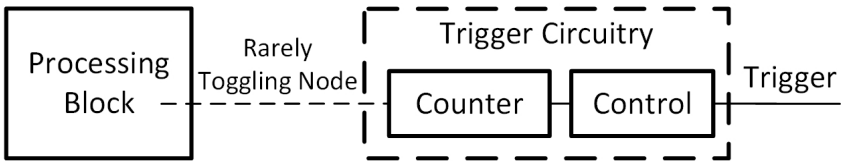


Figure 3.2: An HT trigger circuitry example composed of a rarely toggling node that increments a counter to set the HT trigger.

Such types of larger sized HTs are beyond the scope of our work and, as indicated in Section 2.2.1, can be caught by other HT detection techniques [24–27, 31, 32].

3.2.1 Hardware Trojan Attack Types

In our work, we studied two major classes of the Hardware Trojan threat model described above, namely, HT attacks that target a single point in the architecture (referred to as "single attacks") and HT attacks that target multiple points (referred to as "coordinated attacks").

3.2.1.1 Single Attacks

This type of HT attack targets a single point in the architecture. The goal behind this type of attack is to modify the data either as it arrives at the input of the chip or at the output of any of the internal modules along the data path. The attack attempts to modify the data in the same way as presented in our threat scenario shown in Figure 3.1. Namely, HT trigger circuitry is connected to a logic gate, e.g., an exclusive-or gate, such that when the Trojan is triggered, one bit of the targeted data is complemented resulting in data modification.

Figure 3.3 shows a block diagram of a part of an attacked hardware implementation of the medical device scenario presented in Section 1.2 and Figure 1.2 where ECG and BCG data are captured, processed, encrypted and transmitted. The HT attack shown in Figure 3.3 targets a single point in the architecture (*HT Payload*) by maliciously modifying the input data as it is being sent to the processor. The *HT Trigger* module in the design relies on a logical condition in the hardware encryption unit to be true before triggering the HT.

3.2.1.2 Coordinated Attacks

This type of HT attack attempts to simultaneously target multiple points in the architecture. The HT trigger circuitry is connected to two payloads. The first payload attempts to maliciously modify the data while the second payload tries to hide the effect of the HT so as to pass undetected. For example, an HT could try to simultaneously attack the data and the

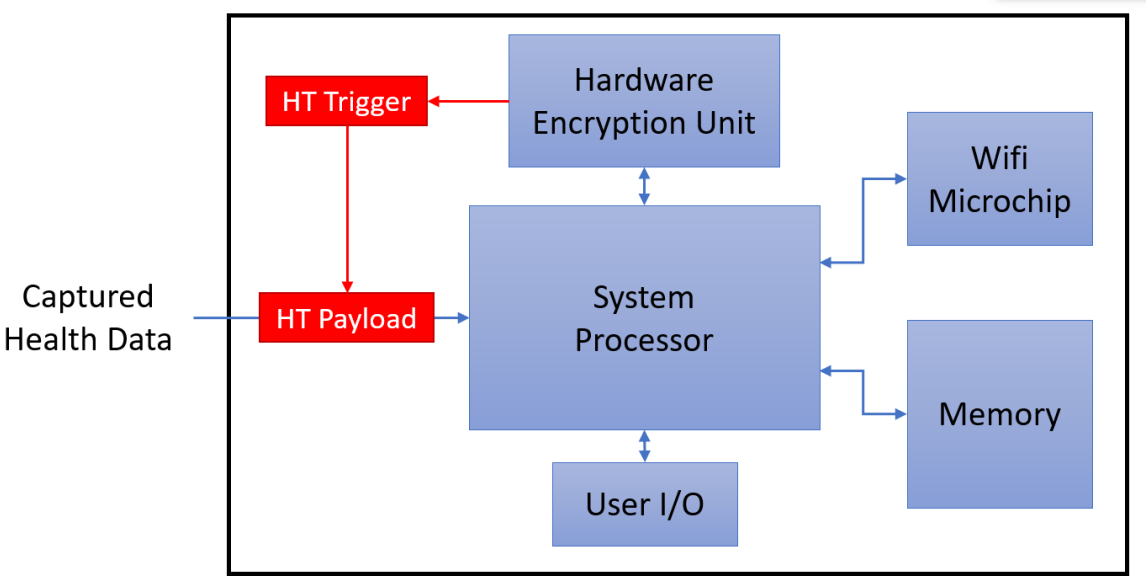


Figure 3.3: An example architecture of medical device hardware showing an HT attack targeting a single point in the design, namely, the input data.

implemented HT detection architecture. If the detection architecture relies on some logic to assert an alarm signal, the HT, for example, could try to suppress such a signal.

Consider an example scenario similar to the one presented in Figure 3.3 with an implemented HT detection architecture as shown in Figure 3.4. The HT detection architecture relies on monitoring the data as it flows in the system to detect any anomalies. Once an anomaly is detected, a signal is sent to the *Wifi Chip* to disable the transmission of the received data. As shown in the Figure 3.4, the HT attack can now consist of an HT trigger that is simultaneously connected to two payloads. *HT Payload 1* performs the same operation of maliciously modifying the data while *HT Payload 2* attacks the decision result of the HT detection architecture. For example, *HT Payload 2* could try to force the decision to always show normal behavior. In this case, even when the HT detection architecture detects that *HT Payload 1* has maliciously modified the data, its decision is being masked by *HT Payload 2* resulting in the attack passing undetected.

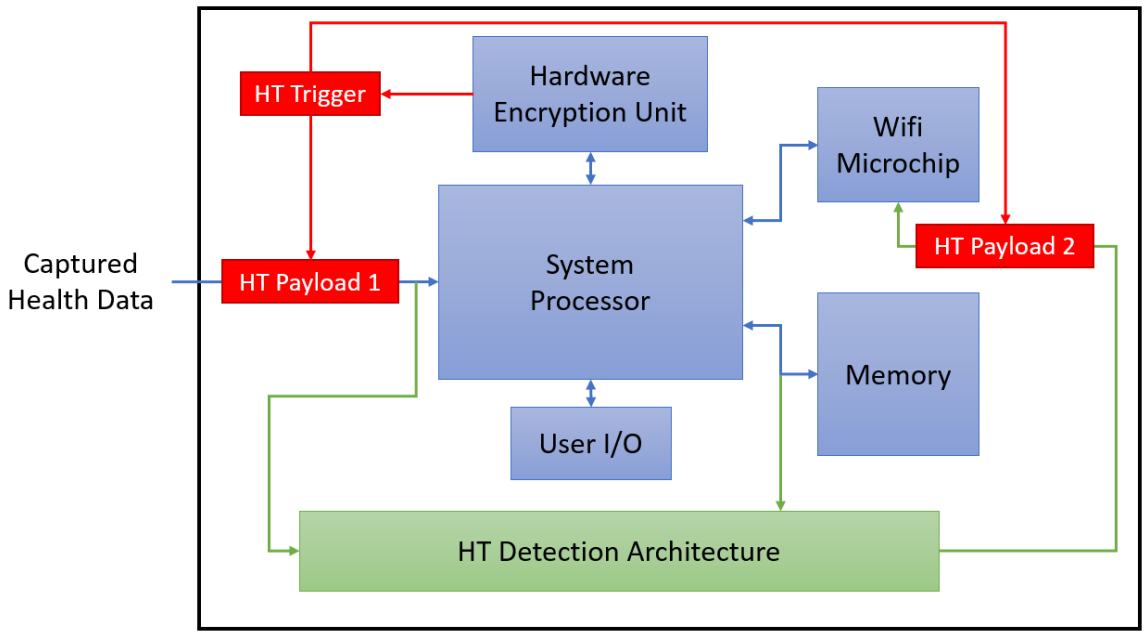


Figure 3.4: An example architecture of medical device hardware including an HT detection architecture showing an HT attack targeting a multiple points in the design.

3.3 Run-time Code Integrity Threat Model

Figure 3.5 presents an overview of the software threat model addressed in this work. Specifically, we focus on malware attacks that target and maliciously change executable code at run-time while the code is present in the memory of an embedded processor. The goal is to detect these types of attacks and assess the integrity of computation performed by the central processing unit of a device to ensure correct behavior of a system.

3.3.1 Run-time Attacks

The software threats on which we primarily focus occur after deployment. Therefore, in our threat model the development environment is assumed to be protected from malicious alterations. In this dissertation, we do not address attacks prior to run-time as there have been a plethora of techniques introduced in the literature to help in protecting compile-time code and allocating secure storage for sensitive information [6, 66].

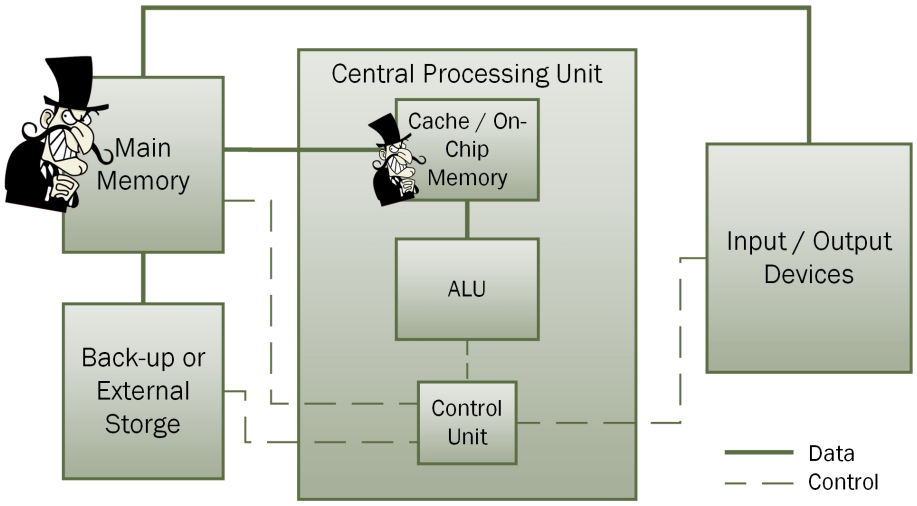


Figure 3.5: An overview of the software threat model showing malware affecting code at run-time while it is present in the memory of a processor.

3.3.2 Code Injection/Modification Attacks

From the different types of common weaknesses and vulnerabilities described in Section 2.3.1, we focus on software attacks that modify user process code or inject new code at run-time. Therefore, attacks that attempt to modify executable memory contents (e.g., via buffer overflow and/or code injection/modification) are primary candidates for the types of vulnerabilities that our work detects. The result of successful code injection attacks can be disastrous. For example, it can result in data loss or corruption, lack of accountability, or denial of access and in extreme cases can lead to complete system takeover.

3.3.2.1 Hollow Process Injection or Process Hollowing

Hollow process injection or *process hollowing* is a variation of the evasive type of code injection/modification malware [67]. The process hollowing technique used by these types of malware utilizes a legitimate process that is loaded on the system to act as a container for hostile code. When the malware is triggered at run-time, it attaches itself to the legitimate process and replaces the process' code with malicious code. Examples of some recent malware that exploited the process hollowing method to inject code into running applications

are the BadNews Android malware [8] and Stuxnet [68]. This evasive type of malware injection constitutes a major focus of our work. Therefore, our attack model covers the case of a malware that inserts itself into a system and tries to maliciously inject code into another process to end up modifying the process's functionality or leak sensitive information. Attacks of this type are typically performed on the text segment of a process address space and can target code down to instruction-level modification. Thus, these types of stealthy malware are hard to detect and prevent since they may be hidden anywhere on the system or may be inserted at run-time to target a specific embedded systems application.

3.4 Threats not Addressed in this Work

The work presented in this dissertation does not address Hardware Trojan (HT) attacks that result in the leakage of sensitive information and/or disabling the functionality of a system [25]. For example, our work does not currently detect an HT attack resulting in the transmission of secretly stored keys in hardware. Similarly, our work does not target HT attacks that try to disable a digital chip by momentarily shutting it down to perform a maliciously desired operation. In addition, HT attacks with a relatively large hardware footprint are not the major focus of our work since there are plenty of techniques presented in the literature, such as side-channel analysis and digital systems test techniques, that cover these types of attacks (refer to Section 2.2.2).

On the software side, the techniques presented in this dissertation do not address the unchanged executable code portion of return-oriented programming (ROP) attacks or code reuse attacks where an adversary modifies the behavior of a program by utilizing existing code in the system to launch their attack; on the other hand, any changes in the executable code – however small – are addressed by this thesis [69, 70]. In addition, our presented and implemented architecture does not currently support Just-in-Time (JIT) compilation where executable code is generated and modified at run-time [71]. Moreover, attacks on the executable processes' page addresses and attacks that end up leaking confidential information

without modifying the pages' contents are currently out of the scope of this work.

3.5 Summary

This chapter presents in detail the different classes of hardware and software attacks that are targeted by our work. Specifically, we focus on extremely small HTs in embedded medical devices which when triggered attempt to modify the functionality of the design. In addition, our work targets software attacks that end up maliciously modifying application code at run-time so as to corrupt the computations performed by an embedded medical device.

CHAPTER 4

HARDWARE TROJAN DETECTION USING HARDWARE SIGNATURES

4.1 Introduction

This chapter presents an embedded systems HT detection architecture that combines multiple novel HT detection techniques [19–22]. Our architecture is motivated by the health monitoring scenario presented in Section 1.2 and targets the HT threat model described in Section 3.2. Specifically, our technique detects HT attacks that are extremely small in size and which, once triggered, attempt to modify the functionality of the chip by attacking the user's health data. The captured ECG and BCG data have known relationships which we take advantage of to create multiple types of signatures that check for the data's integrity at run-time. We embed different techniques of signature generation deep in the hardware (during data harvesting), and then we check for the validity of these signatures during digital processing to ensure that the chip has no HT attacks and that the data's integrity is maintained.

Therefore, this chapter covers novel detection mechanisms used to detect malicious hardware modifications that result in corrupting a user's data as it is being captured in real-time. The rest of this chapter is organized as follows. Section 4.2 presents an overall introduction to our HT detection architecture. Sections 4.3, 4.4 and 4.5 discuss in detail the three different techniques used for signature generation and testing. Section 4.6 discusses combining the three different types of signature generation and testing mechanisms and highlights the advantages of doing so. Section 4.7 presents a modification to our architecture to help detect coordinated HT attacks that target multiple points in a design. Finally, a summary is drawn in Section 4.8.

4.2 Architecture Overview

To target extremely small HT attacks that maliciously modify data as it is captured and processed by digital microchips (see our threat scenario presented in Section 3.2), we devise an architectural level solution where we split our design into two separate chips. Our approach is similar in principal to the concepts presented in [72, 73] where a prover and verifier are used together to verify correct execution on a chip. However, our work relies on checking the integrity of the data as opposed to verifying the full specifications of the hardware components. The premise is that for data integrity there is no need to verify every single specification in a hardware design as long as we can guarantee with a high probability that the actual output of our design is as expected. Figure 4.1 shows an overview of our presented architecture. Our method relies on the idea of generating different types of signatures during data harvesting by sensors and then checking for these signatures later during data processing and encryption.

The left chip in Figure 4.1 represents our first chip or "Chip 1." This chip is responsible for performing analog-to-digital conversion and initial signature generation. Thus, we assume that Chip 1 can use older technologies and can be fabricated in a trusted fab (for example, where employees have security clearances). In our design, we choose to create two types of signatures, a digital signature and an analog-based signature. The two created signatures in Chip 1 along with the captured data are then passed on to Chip 2 in Figure 4.1.

Chip 2 is responsible for data processing, data encryption, and signature regeneration and testing. Specifically, in this work we perform up to three types of signature checks: digital, analog and physiological-based signature testing. Alarm signals are generated out of each of these signature testing mechanisms. The alarm signals are used to inform upper level firmware or software of any potential health problem or hardware attack/error. Chip 2 is assumed to be fabricated using a state-of-the-art process node, usually by another company in a different country, to provide the required complexity and performance needs

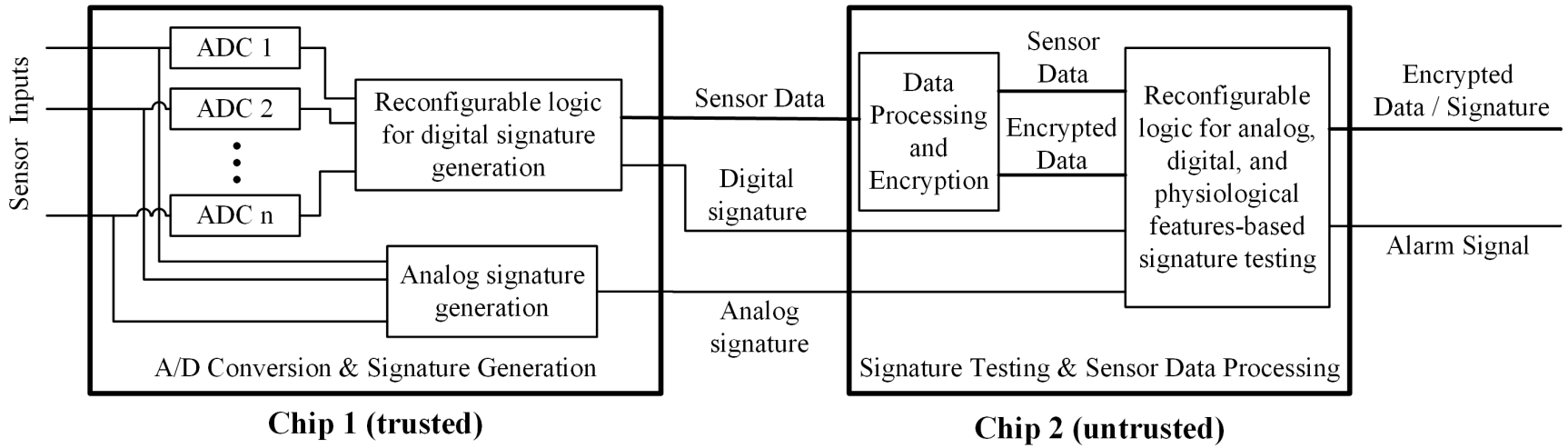


Figure 4.1: An overview of our HT detection architecture showing how a design can be split into two chips. Chip 1 performs signature generation during data harvesting, and Chip 2 checks for these signatures during processing and prior to transmission.

of such a chip. Thus, we consider that an HT can be injected into any part of this chip including the primary inputs. To detect such types of HT attacks, including ones that target the signature testing logic, it was reasonable for us to embed our signature generation and testing mechanisms in reconfigurable logic providing the fabrication company with no information regarding how to insert HTs and allowing for the flexibility in terms of tailoring the testing mechanisms to the application's security needs over the course of the chip's lifetime.

Incorporating signature generation and testing deep in the hardware provides several advantages. First, it enables checking the integrity of the data as early as possible. Second, it improves the security of the overall embedded system design by further complicating the job of a potential malicious entity. Finally, having multiple signature generation techniques – i.e., digital, analog and physiological-based signatures – helps in complicating the job of an attacker and in improving the analysis and decision making of our architecture. In fact, simultaneously issuing multiple alarm signals by three independent types of signature generation techniques reduces the false positives and more importantly the false negatives of our approach.

In the following sections, we introduce and describe the details of each of our signature generation and testing mechanisms. We start by describing the details of the digital signature generation and testing in Section 4.3, followed by the analog signature generation and testing in Section 4.4, ending with the details of the physiological-based signature generation and testing in Section 4.5.

4.3 Digital Signatures

The first type of signature generation and testing mechanism utilized in our architecture is based on a digital form of hardware signatures. As mentioned in Section 2.4, signatures can be generated using different techniques and algorithms (HMACs, CBC-MACs, MISRs, etc.). A selected hardware implementation of one of these signature generation algorithms

is used to compress multiple sets of sensor data to initially create a golden signature and later assess the integrity of the captured data by checking against the created signature. In our work, we choose to implement MISRs as our digital signature generators for their advantage in terms of area and energy consumption over MACs and specifically SHA as shown in Section 2.4. The following subsections describe in detail the part of the architecture responsible for generating the digital MISR signature in Chip 1 and the testing for that signature in Chip 2 to detect malicious hardware modifications.

4.3.1 Chip 1: A/D Conversion and Digital Signature Generation

Figure 4.2 shows a more detailed block diagram of the part responsible for generating the digital signature from the sensor data in Chip 1 or the left chip in Figure 4.1. In this work, we use the BCG force that is along the vertical direction, i.e., the BCG Head-to-Foot (HF) signal, along with the ECG signal. ECG and BCG HF sensor inputs are first passed through analog to digital (A/D) converters before being momentarily stored into First-In-First-Out (FIFO) buffers. The FIFO buffers are 16 bits wide and two slots deep giving the ability for every FIFO to store two consecutive sets of sensor inputs. The output of every FIFO is concatenated to form 32-bit blocks which are in turn concatenated to form a single 64-bit block. The resulting 64-bit block is fed into a MISR [19, 33]. The MISR receives 64 consecutive 64-bit blocks and compresses them into a single 64-bit signature (shown as *Signature 1* in Figure 4.2). *Signature 1* along with the sensor data are passed on to Chip 2.

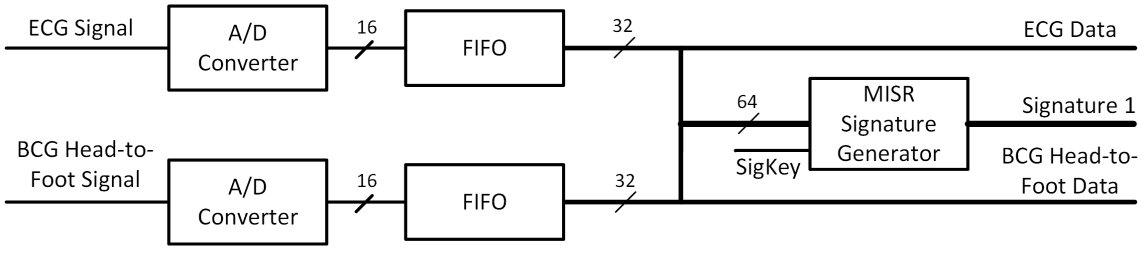


Figure 4.2: A closer look at the logic design that generates the digital (MISR) signature in the first chip.

4.3.2 Chip 2: Digital Signature Testing and Data Processing

Figure 4.3 presents a closer look at the components present in Chip 2 (see Figure 4.1). First, the digital sensor data coming from Chip 1 are concatenated to form a 64-bit block. Second, every block is encrypted using an encryption cipher, such as PRESENT [65], to generate the ciphertext (*CT* in Figure 4.3). Third, the encrypted data is passed through a decryption cipher to regenerate the sensor data plaintext (*PT* in Figure 4.3). *PT* regeneration helps in detecting HT attacks and hardware errors in the encryption and decryption ciphers [19, 33]. Namely, if the *PT* is directly used for signature regeneration, an HT attack on the encryption or decryption cipher will not affect the regenerated signature and thus the attack goes undetected. Fourth, the *PT* is fed through the MISR signature generator to regenerate the digital signature (*Signature 2* in Figure 4.3). Finally, *Signature 1*, the signature coming in from Chip 1, is compared to *Signature 2* to check for any HT attacks or hardware errors in Chip 2. The result of the comparison is fed to a release logic block (right hand side of Figure 4.3) which is responsible for releasing the encrypted data upon success of the comparison. The alarm signal indicates the presence of a potential HT attack or error.

It is important to note that for the signatures that we consider (MISR signatures), we are not aware of known techniques (under the considered threat scenario of extremely small HTs) to quickly compute how a change to an input bit would affect the associated signature. Specifically, since we embed HT detection (MISR signature generation and testing) in re-configurable logic, it appears infeasible for an HT to simultaneously change both the input and the signature in a way that would avoid detection, especially given that the primitive polynomial of the BILBO MISR (Figure 2.3) is not yet placed in the chip hardware logic at fabrication time. Therefore, the MISR signature is accepted to have properties that do not fully satisfy ones of a cryptographic signature. However, if the application requires higher security guarantees, a designer could opt to use a much more secure signature generation algorithm such as a version of the Secure Hash Algorithm at the expense of area and energy consumption.

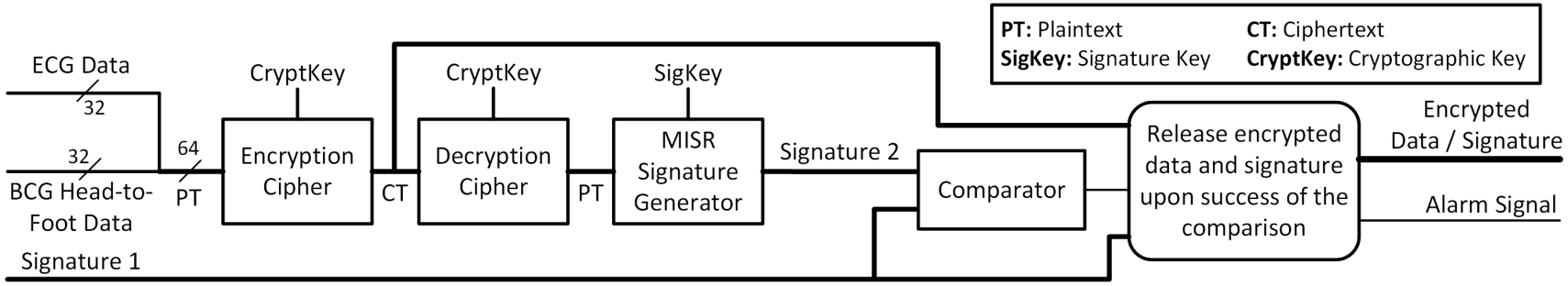


Figure 4.3: A closer look at the logic design that checks for the correct digital (MISR) signature in the second chip.

4.4 Analog Signatures

In addition to compressing multiple ECG and BCG HF samples into a digital MISR signature in Chip 1 of Figure 4.1, we create an analog relationship between every ECG and BCG HF data set sample. We then digitize the created analog signature and pass it along with the MISR signature and the data to Chip 2 in our architecture. The following subsections describe in detail the creation of the analog signature in Chip 1 and the testing of this signature in Chip 2 to better help in detecting HT attacks and unintentional hardware errors.

4.4.1 Chip 1: A/D Conversion and Analog Signature Generation

Figure 4.4 shows an example of a detailed view of the architecture responsible for the generation of an analog signature in Chip 1 in Figure 4.1. First, ECG and BCG HF data harvested from sensors are amplified and filtered to improve the signal-to-noise ratio of the captured data. Second, the filtered data is fed to analog-to-digital converters and prepared to be passed to Chip 2 for encryption and further processing. At the same time, the harvested sensor data are passed through an analog signature generation logic.

In our approach, we create an analog signature using the vector sum (i.e., square root of the sum of the squares of the ECG and BCG measurements) of the captured data as shown in the bottom right-hand side of Figure 4.4. One of the major reasons behind using the vector sum as an analog signature is its importance in analyzing and diagnosing health problems in heart monitoring medical devices [74, 75]. Therefore, we take advantage of the need for such calculations to improve the security by using it to check for the integrity of the captured sensor data. In addition, the simplicity and the abundance of the major components (adders, squarers and square root modules) [76] needed to perform such a calculation make it an intriguing candidate for analog signature generation in embedded medical devices.

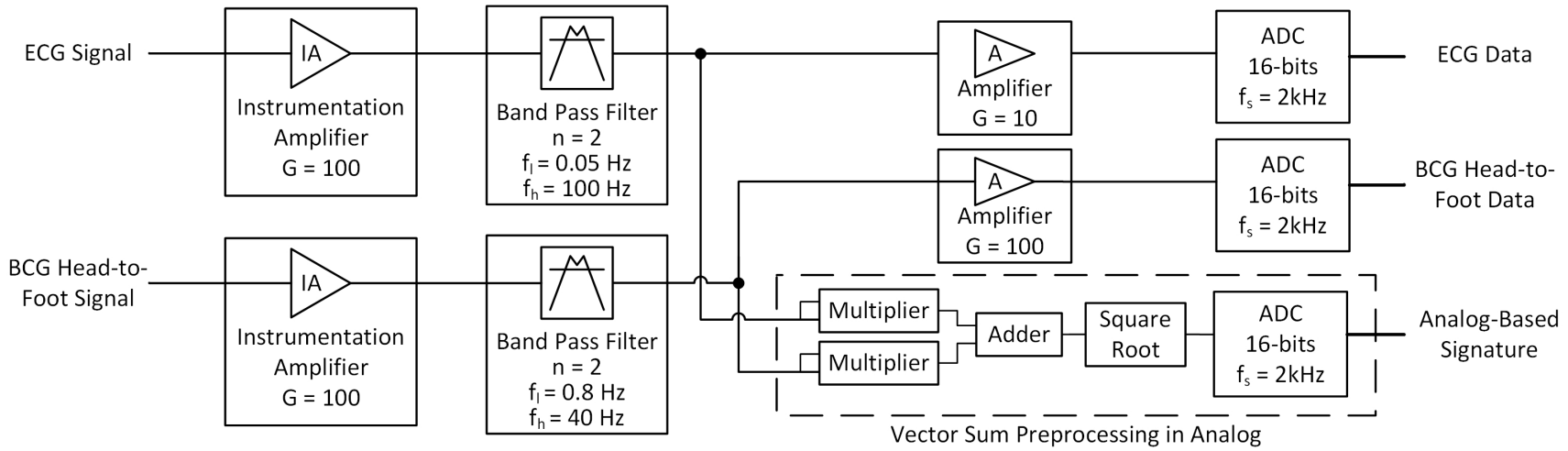


Figure 4.4: A closer look at an example implementation of analog signature (vector sum) generation in the first chip.

Finally, the created analog signature is fed into an analog-to-digital converter and passed on to Chip 2 along with ECG and BCG sensor data.

4.4.2 Chip 2: Analog-based Signature Testing and Data Processing

Figure 4.5 shows a detailed view of the architecture in Chip 2 responsible for the generation of a digital version of the analog signature and its comparison against the digitized golden analog signature coming from Chip 1.

Similar to the process described in Section 4.3.2, the ECG and BCG data coming in from Chip 1 are concatenated to form 64-bit blocks that are in turn encrypted and decrypted using the PRESENT encryption and decryption ciphers respectively. However, the decrypted data in this case are split back into four 16-bit registers. Each two registers hold two consecutive sets of ECG and BCG data. The control unit in Chip 2 schedules the passage of each set of data through the multipliers and adders to regenerate a digital version of the vector sum. Specifically, the first ECG data set is passed through the 16-bit multiplier where the data is squared and saved in a register. Then, the first BCG data set is passed through the multiplier to be squared and then added to the squared ECG data using a 16-bit adder. The generated result is denoted as *Signature 2* in Figure 4.5. It is to be noted that since the analog chip is sampling the BCG data using 16-bit analog-to-digital converters, the 32-bit result of the multiplier is truncated and the most significant 16-bits of the result are used as inputs to the next stage. The loss of precision in generating the analog-based signature using 16-bit multipliers directly affects the application at hand. In our case, the health monitoring application requires an accuracy of only four significant digits after the decimal. Thus, the need for 14 bits to represent the fractional part of the value is enough for this application.

Synchronously, the analog-based signature coming from Chip 1 is passed through a 16-bit multiplier acting as a squarer to generate the squared value of the vector sum (sum of squares of the ECG and BCG data). The result of the squarer is referred to as *Signature 1*

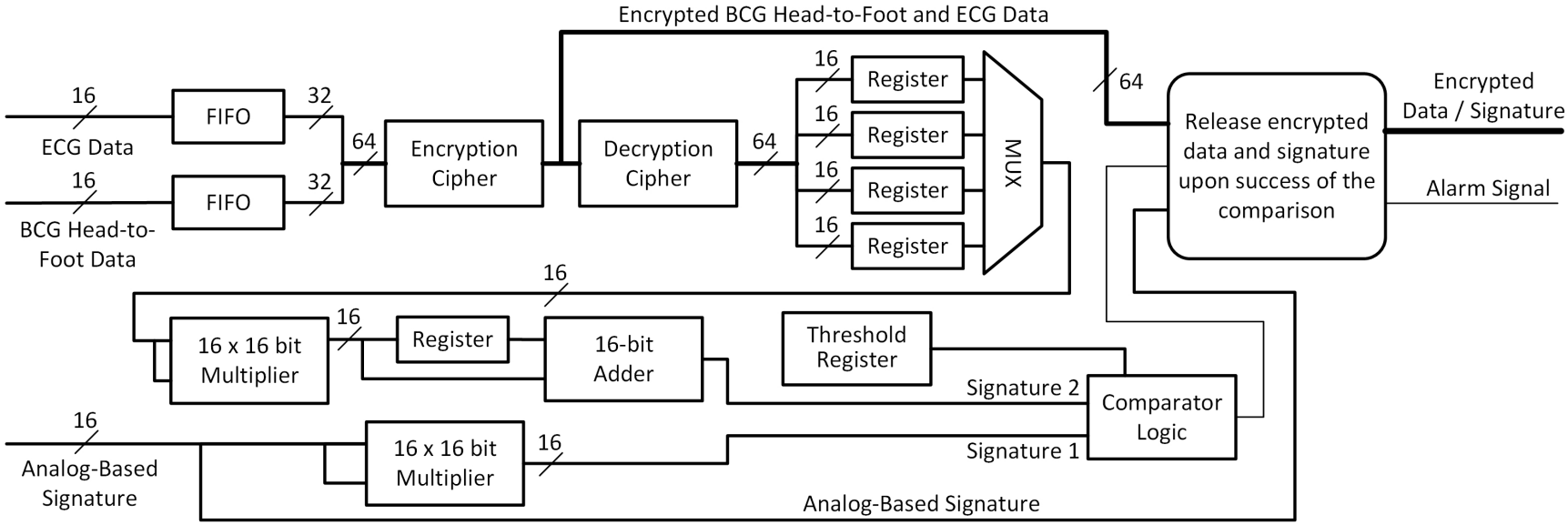


Figure 4.5: A closer look at the hardware of the second chip which is responsible for checking if the analog-based signature matches the digital vector sum calculation.

in Figure 4.5. *Signature 1* and *Signature 2* are then compared to determine if an HT attack or hardware error has happened anywhere in the second chip (Chip 2). Similar to what is shown in Figure 4.3, the result of the comparison is fed into a release logic block to determine whether the encrypted data can be safely transferred out of the chip. If the comparison fails, an alarm signal is raised indicating a potential HT attack or error.

It is to be noted that the comparator logic for the analog-based signature testing, shown in Figure 4.5, behaves in a slightly different fashion than the one used with digital-based signature testing, shown in Figure 4.3. In this case, the comparator logic performs the following:

- $if\ (|Signature1 - Signature2| \leq threshold)$

  declare a match

- $if\ (|Signature1 - Signature2| > threshold)$

  declare a mismatch

where $threshold$ is an input set by the user due to the analog nature of the application and the signature. To reduce the rate of false positives introduced by the analog-based vector sum signature comparison, the threshold register is typically monitored during a learning period for every individual and is adjusted according to the person's data. The threshold is however limited by the accuracy needed by the application at hand. For example, if the application cannot tolerate any difference between the analog-based signature and the regenerated signature, the false positive rate would drastically increase due to the architecture flagging a possible HT attack when in fact the difference is due to the rounding of the analog signature. Thus, for this signature generation technique a compromise has to be made between relaxing the threshold and detecting attacks targeting the least significant bits of the signature.

4.5 Physiological Features-based Signatures

The third type of signature generation and testing discussed in this chapter revolves around extracting physiological features that help in detecting anomalies in users' health data. The extracted features provide some known relationships between the captured data. Our technique utilizes these known relationships to detect HT attacks or unintentional hardware errors in the design.

4.5.1 ECG and BCG Feature Extraction

In this work, we extract two main features from the ECG and BCG HF data, namely, the heart rate and an approximation of the R-J interval [14, 15, 18, 74]. The heart rate can be calculated by finding the heart beat period which is the elapsed time between two consecutive ECG peaks (R-peaks). The R-J interval is defined as the elapsed time between the previous ECG R-peak and the global maximum (J-peak) in the first $400\ ms$ of the BCG HF signal [77]. Since in this work the BCG HF data coming from the force-plate is amplified using an inverting amplifier, the J-peak in such data corresponds to the global minimum found in the same time period.

Figure 4.6 shows a sample recording of the ECG and BCG HF signals. Figure 4.6 also marks the features that we use in our architecture. In fact, the hardware needed to extract these features can be considered part of the design since these features are often used to analyze a person's physiological health [14, 15, 18, 78].

4.5.2 Hardware Peak Detection

The hardware implementation of our physiological-based signature generation technique requires accurately extracting features and relationships between the ECG and BCG HF data. Specifically, an important part of the feature extraction circuitry relies on accurately detecting peaks in the data as it flows through our architecture; thus, we implemented a

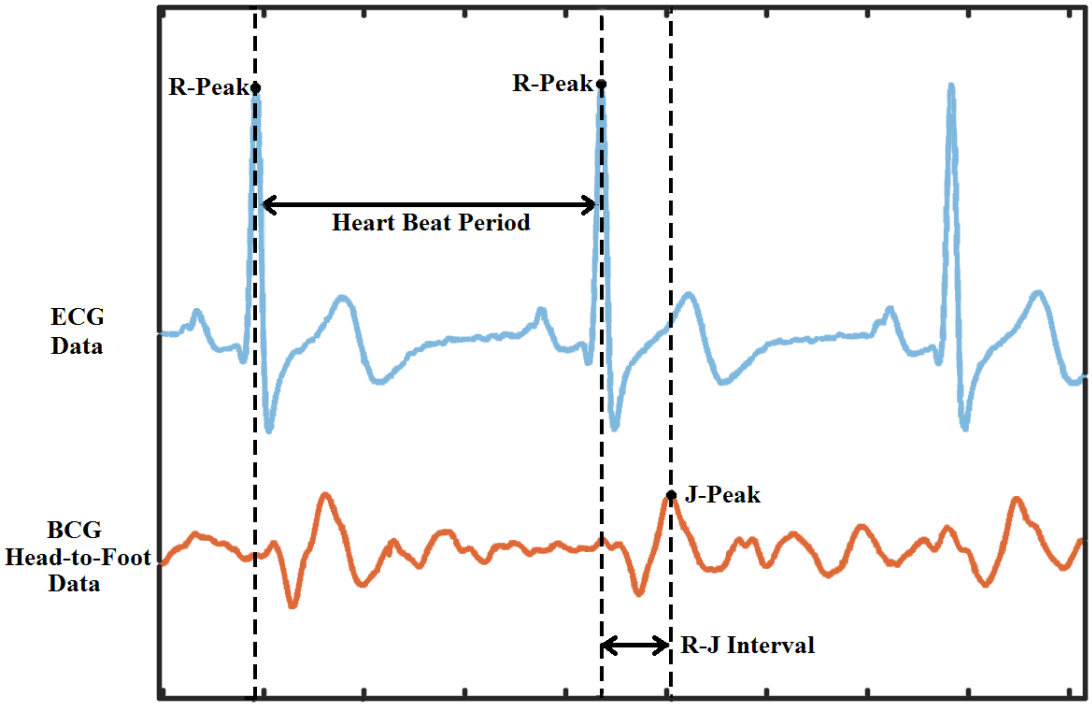


Figure 4.6: Using ECG and BCG Head-to-Foot (HF) Data to extract a person's heart rate and R-J interval values.

hardware efficient and accurate peak detector developed by Jordanov and Hall [79].

Figure 4.7 shows a block diagram of the digital peak detector that we use. This digital peak detector combines noise immunity and speed of peak detection. It operates in two modes: (i) tracking maximum and (ii) tracking minimum. A maximum value is stored when transitioning from tracking maximum mode to tracking minimum mode. Similarly, a minimum value is captured when transitioning from tracking minimum mode to tracking maximum mode.

The peak register (*PREG*) in Figure 4.7 is used to track and capture the minimum/maximum value. The peak detection is controlled by the output value of the *comparator*. Since ECG and BCG data are captured by sensors, the signals are typically coupled with noise. Thus, to improve the accuracy of the peak detection, a noise threshold value is set in the architecture. The *noise threshold* register holds the acceptable noise theshold value for the application at hand. It is used along with its negated value to improve the accuracy by

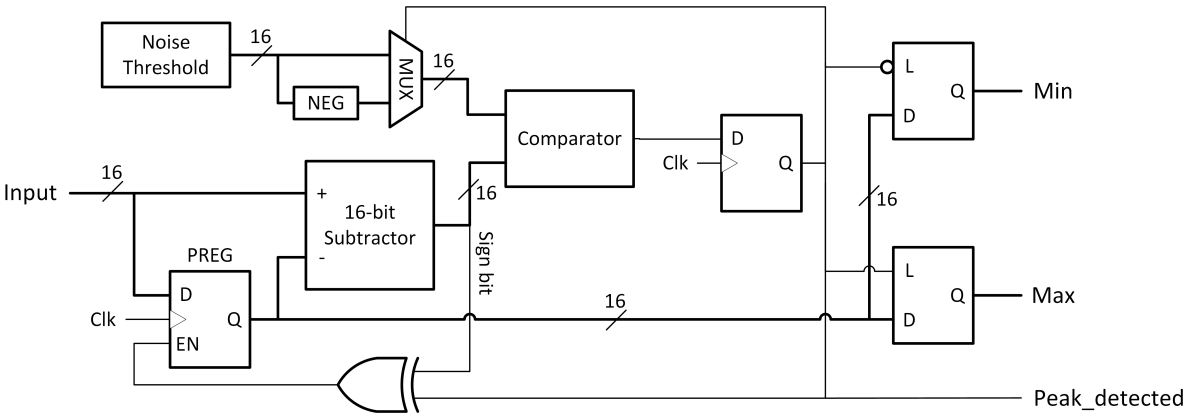


Figure 4.7: Peak detection circuitry with noise immunity and fast detection of maxima and minima [79].

eliminating peaks that are lower than the threshold. This is established by the use of the *16-bit subtractor* and the *comparator*.

On the one hand, when the peak detector is operating in the tracking maximum mode, the *PREG* value is latched in the *Max* latch. On the other hand, when the peak detector is operating in the tracking minimum mode, the *PREG* value is latched in the *Min* latch. Finally, the sign bit of the subtractor's output exclusive-ored (XOR) with the output of the comparator act as the enable signal of the peak register (*PREG*) allowing for proper tracking of minimum and maximum values [79]. The tracked minimum and maximum values will be used in our architecture to extract the different physiological features and relationships between the ECG and BCG data.

4.5.3 Physiological Feature Extraction Hardware

As opposed to the digital and analog based signature generation and testing techniques, the physiological-based signature generation and testing technique does not require a signature generation phase in Chip 1 in Figure 4.1. Instead, this technique extracts features (signatures) only in Chip 2 and relies on the known physiological properties of these features to check for anomalies in the circuit that affect the integrity of the data.

Figure 4.8 shows a detailed view of the architecture responsible for generating and

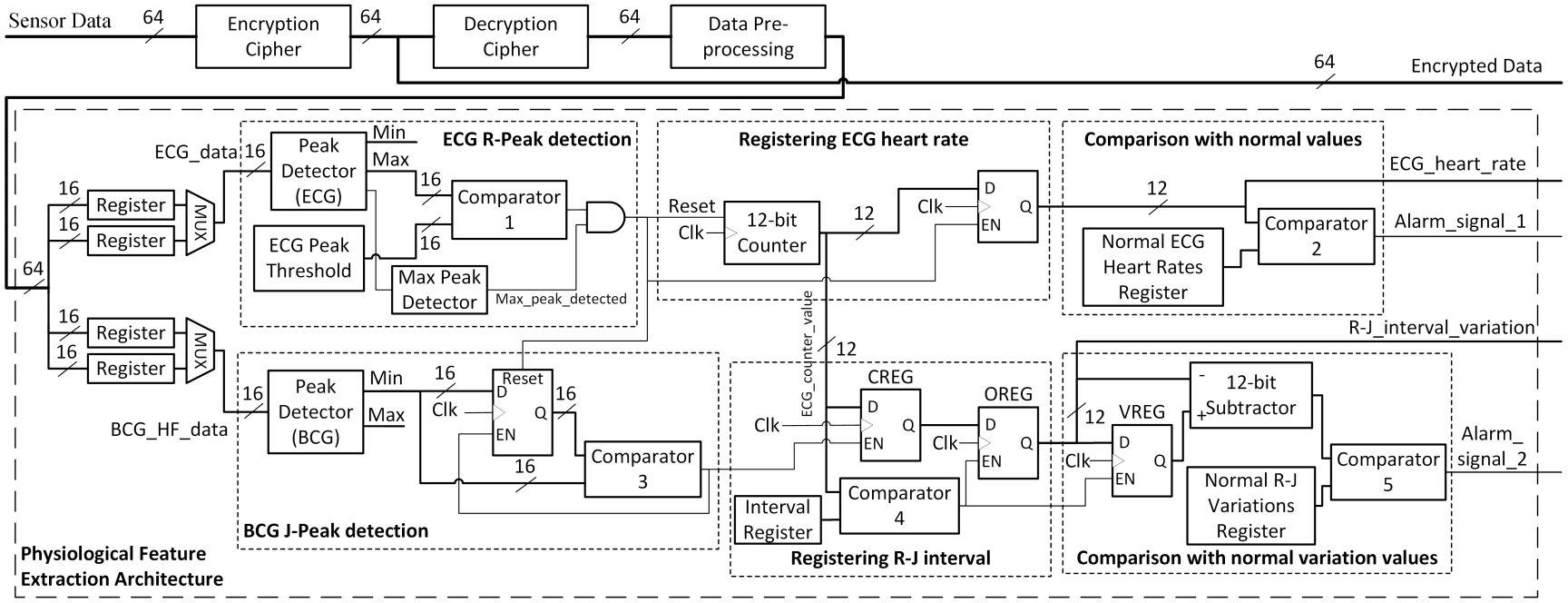


Figure 4.8: Hardware implementing two physiological features extraction and testing. Namely, the heart rate and the R-J interval of an individual are being computed in real time and compared to known normal values.

testing for the physiological-based signatures in Chip 2 of our design. As with the previous two techniques, sensor data (ECG and BCG HF data) coming into Chip 2 are first encrypted and decrypted using the PRESENT encryption/decryption cipher. Then the decrypted data is pre-processed to improve its signal-to-noise ratio for a better feature extraction. The concatenated sets of ECG and BCG data are split back into sets of 16-bit signals. The ECG and BCG data sets are then used to extract the two main physiological features, namely, the heart rate and the R-J interval [14, 15, 18, 74, 77].

4.5.3.1 Heart Rate Extraction

The ECG heart rate is generated as follows. First, the *ECG_data* is passed through the peak detector described in Section 4.5.2 and shown in Figure 4.7. The peak detector outputs a maximum or a minimum value whenever a peak is detected. It also outputs a signal (*peak_detected*) indicating if a peak is detected. Second, this signal is passed through a Max Peak Detector logic block to validate that the detected peak is a maximum and not a minimum. When this is the case, the *Max_peak_detected* signal is set. Simultaneously, the *Max* value output by the peak detector is compared (Comparator 1) with a predefined ECG Peak Threshold stored in a register. If the *Max* value is higher than the threshold, the comparator's output is set to a logic '1' value. Third, the comparator's output is ANDed with the *Max_peak_detected* signal to create a reset for a 12-bit counter. The 12-bit counter is used to count the number of samples between two consecutive R-peaks of the ECG signal. Finally, whenever an R-peak is found, the output of the AND gate is set to one resulting in the storage of the value of the counter in a register and the reset of the counter back to zero. The value stored in the counter denotes a heart beat period which is a representative of the measured heart rate.

The generated ECG heart rate is compared (Comparator 2) with normal values of ECG heart rates to generate and set an alarm signal when an anomaly is detected.

4.5.3.2 R-J Interval Extraction

The R-J interval is extracted as follows. The *BCG_HF_data* is passed through an instantiation of the peak detector to look for minimum values in the data. The *Min* value generated by the peak detector is connected to a register and Comparator 3 to store the lowest *Min* value found so far. This is done by connecting Comparator 3's output to the register's enable signal. The comparator's output is set to a logic '1' whenever the new *Min* value is lower than the previously stored *Min* value in the register. The register's reset signal is connected to the output of the AND gate (from the ECG part of the chip) so that the register is reset whenever an ECG R-peak is encountered.

Comparator 3's output in Figure 4.8 is also used to store the counter value coming from the ECG part of the chip in the register labeled "CREG." This will lead to the storage of the interval between an R-peak in the ECG signal and the current global minimum in the BCG HF signal. To obtain the correct representative value of the R-J interval, the stored value of the counter (in CREG) is not used unless it is confirmed to be the global minimum within the first $400\ ms$ of the BCG HF signal. To perform this function, Comparator 4 is used to compare the *ECG_counter_value* to a predefined interval register value representing $400\ ms$. Once the ECG counter reaches the interval register's value, Comparator 4's output is set and the R-J interval is stored in the output register (OREG).

To monitor the variation in the R-J interval values, a second register (VREG) is used such that the two registers, OREG and VREG, contain the current and the previous values of the R-J interval respectively. The values of these two registers are subtracted using a 16-bit subtractor to determine the variation between consecutive values. In a normal scenario, this variation should not exceed a predefined value [74, 77]. Comparator 5 uses preset values found in the Normal ECG to BCG Rate Variation register to check for this phenomenon and triggers an alarm whenever this phenomenon is violated.

4.5.4 Alarm Signal Severity

As shown in Figure 4.8, the physiological-feature based signature generation and testing architecture generates two alarm signals, namely, *Alarm_signal_1* and *Alarm_ signal_2*. The nature of these two alarm signals differs from the ones generated by both the digital-based signature testing architecture and the analog-based one described in Sections 4.3 and 4.4 respectively. The physiological-based alarm signals indicate one of three status values: "no anomaly," "anomaly" with high fidelity, and possibility of an anomaly which is referred to in our work as the "gray zone" [20].

Determining the severity of the alarm signals depends on the different ranges of the extracted features based on physiology and as reported in the literature [77, 80]. However, it is to be noted that varying these ranges affects the rate of false positives and false negatives generated by our architecture. The decision of which precise range values to place in our architecture is up to the user. Determination of range values is beyond the scope of this work.

The "anomaly" with high fidelity is set when the comparison of the generated feature's value with the normal predefined values shows a large mismatch. For example, *Alarm_signal_1* indicates an anomaly when the value of a resting heart rate goes below $30\ bpm$ (beats per minute) or beyond $130\ bpm$ [80]. Similarly, *Alarm_signal_2* indicates an anomaly when the variation in the R-J interval values exceeds 50% [18]. The no anomaly status is set when the generated feature's value falls within the preset normal range of values. For example, if the calculated heart rate for a resting person falls between $45\ bpm$ and $110\ bpm$, *Alarm_signal_1* declares a "no anomaly" status [80]. Similarly, *Alarm_signal_2* reports a "no anomaly" status when the variation in the R-J interval values is below 15% [18].

If the comparison of the generated feature's value with the predefined normal values falls in a range between the "anomaly" and "no anomaly" levels, the alarm signal declares a "gray zone" status. For example, *Alarm_signal_1* indicates a "gray zone" status when the resting heart rate of a person falls between $30\ bpm$ and $45\ bpm$ or between $110\ bpm$ and

$130\ bpm$ [80]. *Alarm_signal_2* declares a "gray zone" status when the variation in the R-J interval values is between 15% and 50% [18].

4.6 Combining Digital, Analog and Physiological Based Signatures

One of the important aspects of our design is its complimentary nature where it can be applied in parallel with other HT detection techniques. In this section, we show that the techniques introduced in Section 4.2 can be all combined together into one architecture to not only detect HT attacks and hardware errors in medical devices, but also distinguish them from health problems. Similarly, other types of signature generation techniques can be seamlessly integrated into our architecture if needed.

The modified block diagrams of Chip 1 and Chip 2 of our overall design are shown in Figure 4.9 and Figure 4.10 respectively. Figure 4.9 shows that Chip 1 has to be slightly modified to combine both the analog and digital initial signature generation. Specifically, FIFO buffers are introduced to store multiple sets of data. The stored sets of data are then passed through a BILBO MISR block to create the MISR-based signature (digital signature) which is passed along with the analog-based signature (vector sum of the ECG and BCG HF data) to Chip 2.

Figure 4.10 shows the detailed architecture of Chip 2 where the signatures are checked to verify the integrity of the data before transmission. First, the ECG and BCG HF data are encrypted and decrypted; then, the resulting plaintext is used to generate all three types of signatures, the digital, analog and physiological-based ones. Thus, the release logic is now controlled by multiple alarm signals coming from two types of signature comparators (*Comparator Logic 1* and *Comparator Logic 2* in Figure 4.10) and from the heart rate and R-J interval feature extraction circuitry. *Comparator Logic 1* in Figure 4.10 validates the digital-based signature (MISR), and *Comparator Logic 2* validates the integrity of the analog-based signature (vector sum).

It is important to note that adding multiple types of testing for the integrity of data is

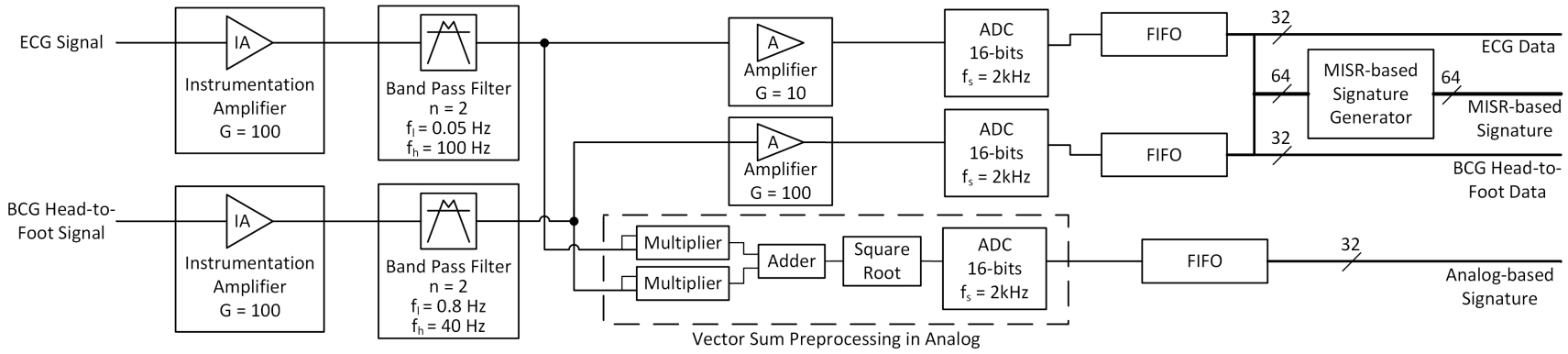


Figure 4.9: Final implementation of the first chip (Chip 1) showing the generation of both digital and analog signatures during the data harvesting process.

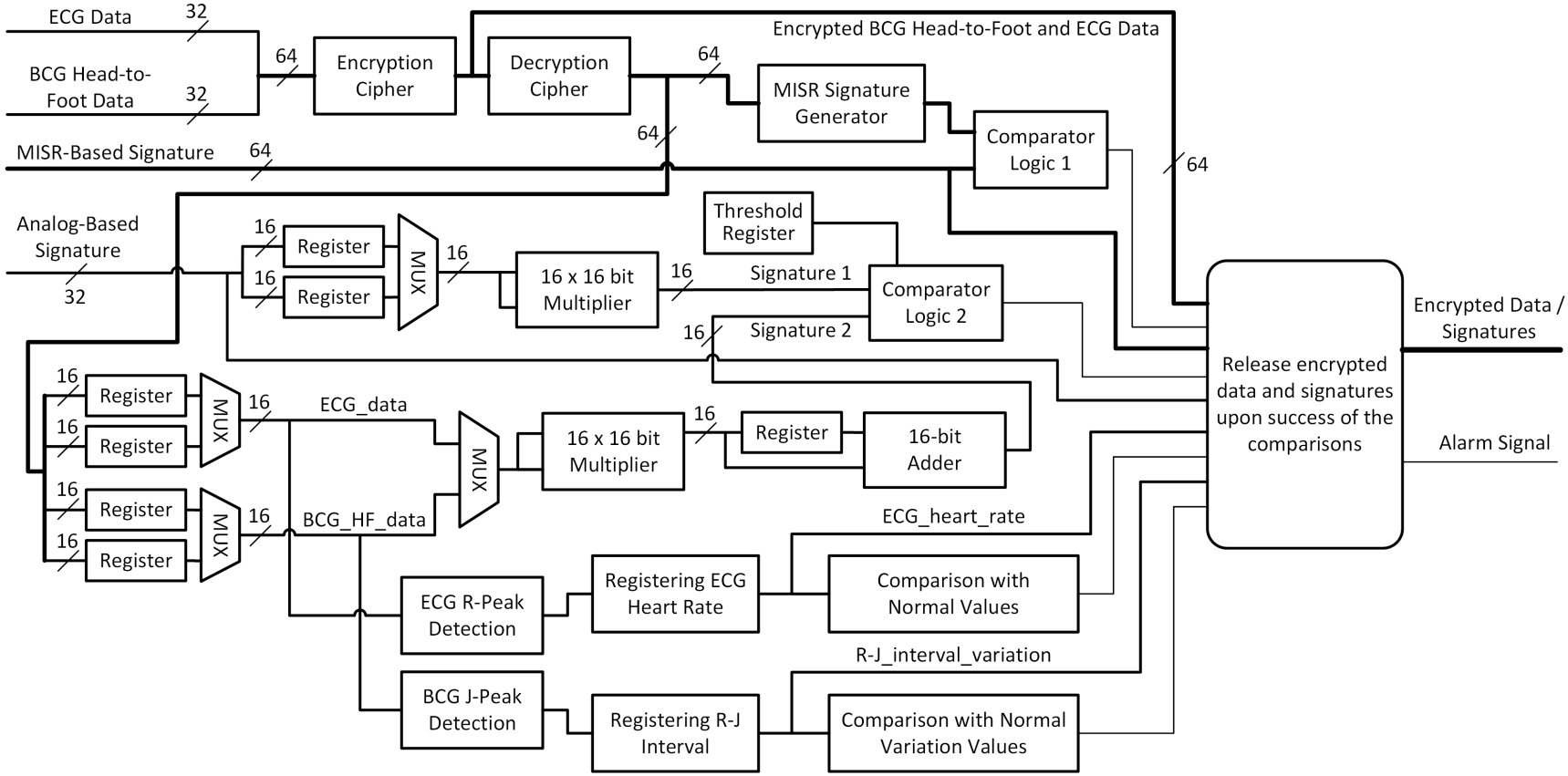


Figure 4.10: Final implementation of the second chip (Chip 2) showing the regeneration and testing of digital, analog and physiological features-based signatures.

important for specific types of HT attacks and increases the security level of the architecture by allowing for smarter decisions when analyzing the results of multiple signature generation techniques. In addition, the incorporation of signature comparison (for both digital and analog-based signatures) along with physiological feature extraction helps in distinguishing HT attacks or other hardware errors from health problems.

To better show the analysis of the different alarm signals of Figure 4.10, we present in Table 4.1 some plausible decisions that could be taken upon looking at the combination of all alarm signals in the architecture. For example, an HT attack modifying low order bits of the BCG HF data and the analog-based signature might go undetected in the architecture shown in Figure 4.5. However, for an architecture where even detection of such changes to low order bits are needed, adding the MISR-based HT detection approach may be desirable, especially if the digital systems test structures already include BILBO logic which can be reconfigured for use in HT detection. Adding the MISR-based HT detection would help in detecting an attack on any of the data bits as the authors are unaware of any publication that shows a way of simultaneously modifying the data bits and the MISR-based signature without being caught [19, 33]. As shown in Table 4.1, even when the analog-based signature module indicates a match, if both the digital and physiological-based signature modules indicate a mismatch, the architecture can then guarantee that an anomaly is present in the system.

Another example that benefits from the incorporation of the analog, digital and physiological-based approaches is the case of an HT attack on the ECG and BCG HF data inputs in the architecture of Figure 4.10. Suppose the HT were to consist of some minimal logic to swap the values of the ECG and BCG HF signals. This type of specific attack will go undetected by the sole use of the analog-based signature detection approach. However, since the MISR-based signatures are created by compressing each ECG and BCG HF data component independently and not by generating a relation between the two components, the attack will be detected and flagged by the combined architecture as shown in Table 4.1.

Table 4.1: Analysis of combining digital, analog and physiological-based signature generation and testing

		Digital-based Signature			
		Match		***Mismatch***	
		Analog-based Signature		**Analog-based Signature**	
		Match	***Mismatch***	***Match***	***Mismatch***
Physiological-based Signature	***No Anomaly***	Correct operation	Possible attack on the analog signature generation architecture	Possible HT attack or hardware error in low order bits	Possible occurrence of a false negative by the physiological-based signature architecture
	Gray Zone	Potential health problem (person is recommended to seek medical help)	Potential health problem or possible attack on the analog signature generation architecture	Possible HT attack or hardware error along with a potentially minor health problem	Hardware Trojan attack or hardware error
	Anomaly	Person is asked to seek immediate medical help	Possibility of a serious health problem or attack on the analog signature generation architecture	HT attack or hardware error along with a possibility of a serious health problem	Hardware Trojan attack or hardware error

Finally, as Table 4.1 shows, the combination of the three techniques not only improves the decision making in whether an anomaly is present in the circuit, but also helps in identifying whether the anomaly is due to an actual hardware attack/error or due to a health problem that the individual is having. For example, if both the analog and digital-based techniques indicate a match while the physiological-based feature extraction circuitry indicates an anomaly, the person is asked to seek immediate medical help indicating a high possibility of a serious health problem.

It is important to note that in this work our architecture focuses only on creating means of detecting HT attacks, hardware errors and/or health problems by asserting alarm signals when the respective conditions are believed to be present. The decisions and countermeasures to such types of attacks or errors are kept to be processed and analyzed by higher level policies and protocols.

In addition, since our techniques rely on the correctness of signature generation and comparison, it is worth mentioning the chip aging effects on the reliability of our approach. The analog-based vector sum and the physiological-based signature generation architectures already implement threshold registers due to the inherent variations in the values of these signatures. These threshold register values can be calibrated as the chip ages to compensate for signature changes over time. However, the MISR-based signature generation presents a more challenging scenario since single bit flips would generate false positives. Techniques used to address the aging effects in hardware implementations of hashing algorithms and PUFs, such as the use of error detection and correction, could be added to our architecture to compensate for chip aging [81].

4.7 Defending Against Coordinated Attacks

The architecture presented in Figure 4.10 does not detect all types of coordinated HT attacks. For instance, consider an example of a coordinated HT attack targeting our developed architecture as shown Figure 4.11. The HT's trigger circuitry is connected to two

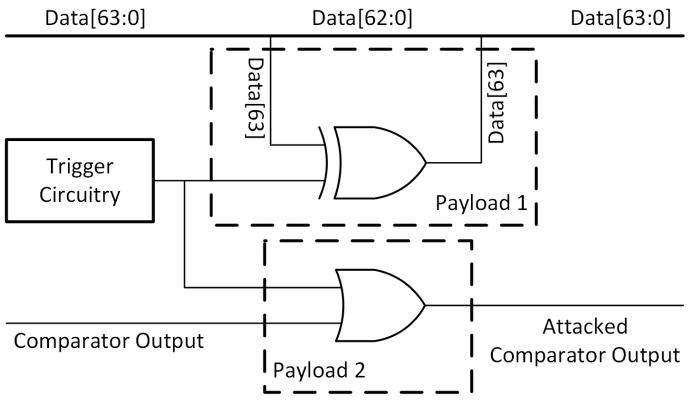


Figure 4.11: An HT attacking both an internal data bus in the design and the output of a comparator.

payloads. The first payload (Payload 1 in Figure 4.11) affects the architecture in the same way as discussed earlier in our threat model in Section 3.2 as well as in Figure 3.1. The second payload (Payload 2 in Figure 4.11), which is triggered at the same time, attempts to set the output of the comparator logic to a fixed value indicating a matched comparison. This way, even if the data is altered resulting in an altered regenerated signature, the comparator will still indicate a success in the comparison and the attack will go undetected.

To prevent such a type of attack, for every comparator in the design we insert a module in reconfigurable logic, the *comparator testing logic* block (Figure 4.12), to specifically

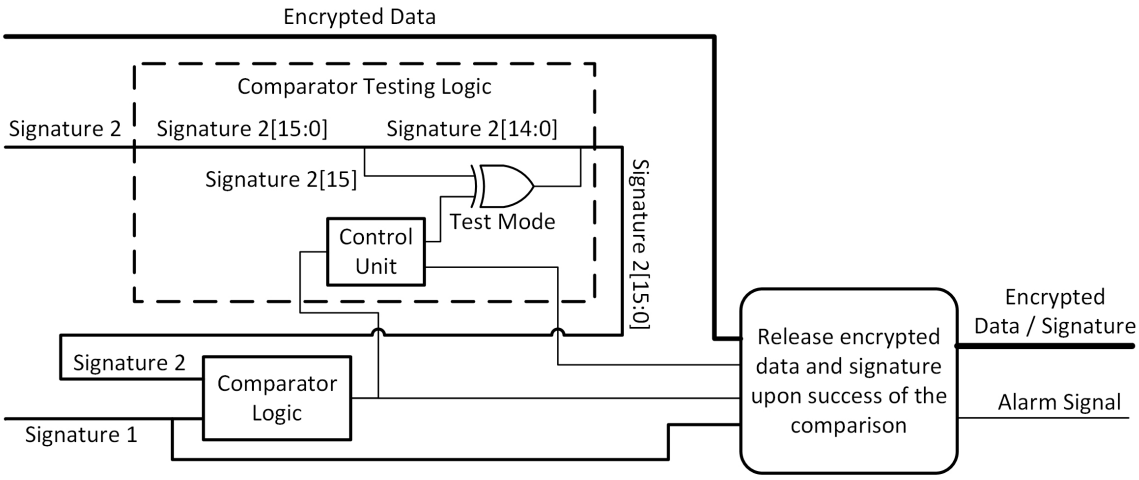


Figure 4.12: A comparator testing logic unit inserted to verify the correct operation of a comparator and detect any HT attacks that attempt to alter the comparator's result.

check for the attack's effect on the comparator. The comparator testing logic periodically checks for the expected behavior of the comparator by asserting a *test mode* signal as shown in Figure 4.12. In this way, the regenerated signature is intentionally modified and the result of the comparison is checked. If the *comparator* indicates a match, then we know with a high fidelity that the *comparator* is under attack. When the *test mode* signal is deasserted, the regenerated signature is passed without any alteration and the circuit behaves normally. It is worth mentioning that for energy constrained embedded and medical systems, such as battery-powered devices, extensive testing might not be feasible and thus the comparator testing logic might be simplified or completely removed. An alternative could be to directly embed the comparators in reconfigurable logic to strengthen the architecture against HT attacks that attempt to provide a workaround against our designed signature checking technique.

4.8 Summary

In this chapter, we present a novel two-chip architecture for detecting malicious hardware modifications at run-time in medical devices through the use of hardware signatures. Specifically, our architecture is designed to detect extremely small HTs in embedded medical devices which when triggered attempt to modify the functionality of the design. Three different techniques for signature generation are developed, namely, analog, digital and physiological-based signatures, by taking advantage of known relationships between health sensor data. In addition, we present a modification to our designed architecture to detect coordinated HT attacks that target multiple points in a design.

CHAPTER 5

RUN-TIME CODE INTEGRITY ARCHITECTURE

5.1 Introduction

In this chapter, we present a hardware-assisted technique to detect malicious software activities that end up modifying executable code at run-time [23]. Specifically, our technique performs binary code analysis and page-based signature (hash) generation of critical applications running on an embedded system to ensure the integrity of computation performed by a medical device. We perform run-time memory monitoring through a separate and isolated hardware monitor that performs on-the-fly page-based hash generation and testing. Malicious modifications to running executable code are rapidly caught and flagged, e.g., to the operating system (OS), indicating the presence of abnormal behavior.

The remaining sections of this chapter are divided as follows. Section 5.2 presents our approach and methodology. Details of our implementation and target architecture along with the associated challenges are presented in Section 5.3. In addition, Sections 5.2.3 and 5.3.2 show how our architecture modification for assessing kernel-level process integrity on top of user-level process integrity provides a more robust implementation and guarantees that attacks on critical kernel-level modules are detected in real-time. Finally, conclusions are drawn in the summary in Section 5.4.

5.2 Overall Approach and Method

To protect a process running on an operating system such as Linux, we present an architectural approach composed of a hardware monitor that is tightly coupled with the physical memory of a processor as shown in Figure 5.1. Our approach involves generating hashes of the monitored process's executable pages at compile-time and storing them in a secure

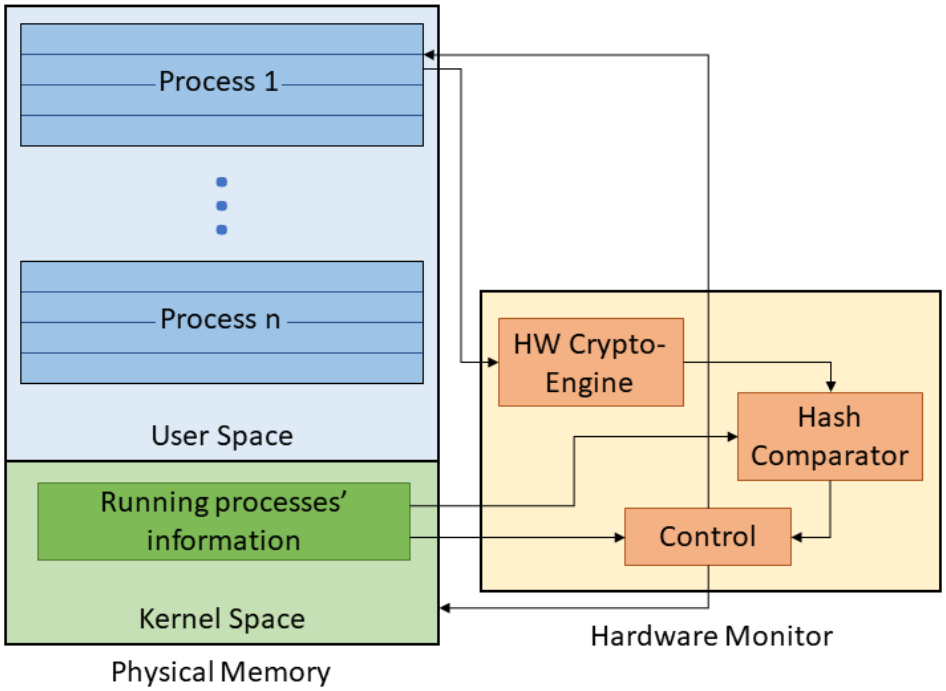


Figure 5.1: Our novel process integrity approach showing a hardware monitor tightly coupled to a processor's physical memory.

location to be later compared to run-time generated hashes during process execution. Our approach aims to provide a dynamic way of assessing system process integrity while maintaining isolation from software and corresponding software vulnerabilities.

5.2.1 Detailed Approach

Figure 5.1 shows a conceptual view of our hardware/software codesign approach implemented at the memory interface with the hardware modules used in our architecture to capture evidence of malicious code execution at run-time. We tightly couple the hardware monitor to the physical memory of a processor to periodically perform memory probing through page-based code analysis. After the compile-time generation of the pages' golden hashes, the kernel informs the hardware monitor once an application is scheduled to run on the processor. The hardware monitor communicates with the kernel to extract the desired running process's information and state. In addition, the monitor is given full access to the

physical memory where it is able to read the contents of the process's loaded pages. In our work, we focus only on executable pages; therefore, data segments and unmapped regions within a page are zeroed out before any compile-time or run-time hash generation happens. It is important to note that the hardware monitor only requires read access to the memory, and thus if writes are disabled it is not possible to corrupt the memory through hardware probing.

5.2.2 Methodology

To better understand the overall process starting from the preparation of the golden hashes to the verification of code integrity at run-time, we present the flow of our technique in Figure 5.2. The overall method is divided into two general phases, (i) a compile-time phase and (ii) a run-time phase.

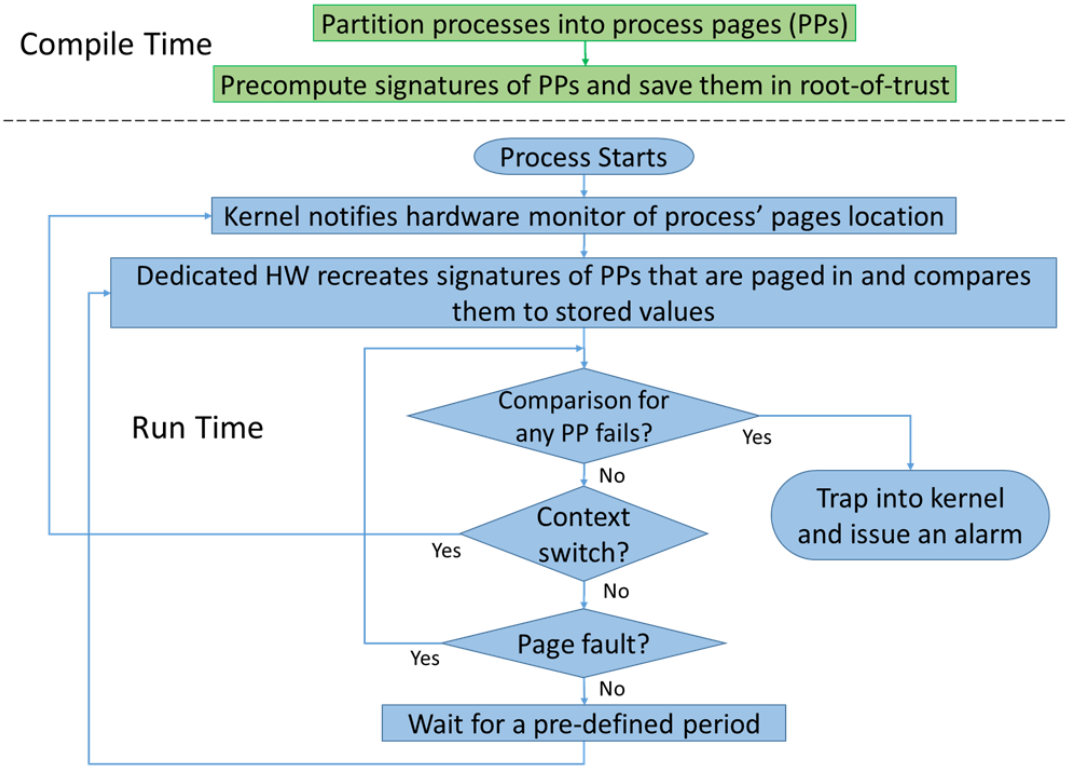


Figure 5.2: The flow of our presented approach where golden hashes of process pages are generated at compile-time and then checked during run-time to verify the integrity of the running process.

During the compilation phase and after the target process has been compiled into its equivalent binary, the executable code of the process is partitioned into pages as defined in its Executable and Linkable Format (ELF) file [82]. The content of every page (4 KB) is hashed using a secure cryptographic algorithm to generate a golden signature of the page (in the case of the Secure Hash Algorithm SHA256, the generated hash is a bitstream of 256 bits). Due to the avalanche effect provided by the cryptographic algorithm, any bit change in the page content will result in a significant change in the generated hash of that specific page. The generated hashes are stored and indexed into a database in a secure and trusted location (e.g., a software or hardware root-of-trust [66]).

At run-time, when the process is loaded into memory, the kernel notifies the hardware monitor of the process pages' locations in memory. This is accomplished by inserting a monitoring kernel process that extracts the memory maps of the target process and translates the virtual addresses of the pages to their corresponding physical addresses and page frame numbers (PFNs) before sending them to the hardware through a communication port set up between the hardware monitor and the kernel. Once the hardware is aware of the locations of the process's executable pages, the monitor grabs the content of every page utilizing a direct memory access (DMA) controller. The content of each page is then passed through a hardware (HW) crypto-engine, and a run-time hash of the page is generated. The newly generated hash is compared to the same page's golden hash as retrieved from secure storage. Currently, the comparison is performed by string matching. However, if faster implementations are required, it is possible to consider techniques that only look for similarities between the golden and regenerated hashes especially since a single bit change in the page contents will result in a significant change in the regenerated hash. The process of regenerating and comparing hashes happens in the hardware monitor and thus is immune to any of the software attack types defined in our threat model (see Section 3.3).

To allow for continuous monitoring, the hardware monitor's control unit periodically restarts the operation of hash regeneration and comparison once all the process pages have

been checked. In addition, to allow for the protection of multiple applications from malicious code modification, the hardware monitor also checks for OS context switches. If the OS starts the execution of any one of the monitored processes, the hardware monitor will similarly grab the code contents of the newly scheduled process, one page at a time, and check for the corresponding pages' integrity. If at any point the comparison between the run-time regenerated hash and the stored golden hash of any page fails, the hardware monitor triggers an alarm by trapping into the kernel to indicate an integrity violation in the running application.

5.2.3 Kernel Process Integrity Extension

Our approach is scalable and can be expanded to provide assessment of kernel-level process integrity. In that scenario, a dedicated memory has to be attached to the hardware as shown in Figure 5.3, providing an embedded hardware root of trust. In addition, the dedicated hardware monitor is allowed to interface with and access the kernel address space

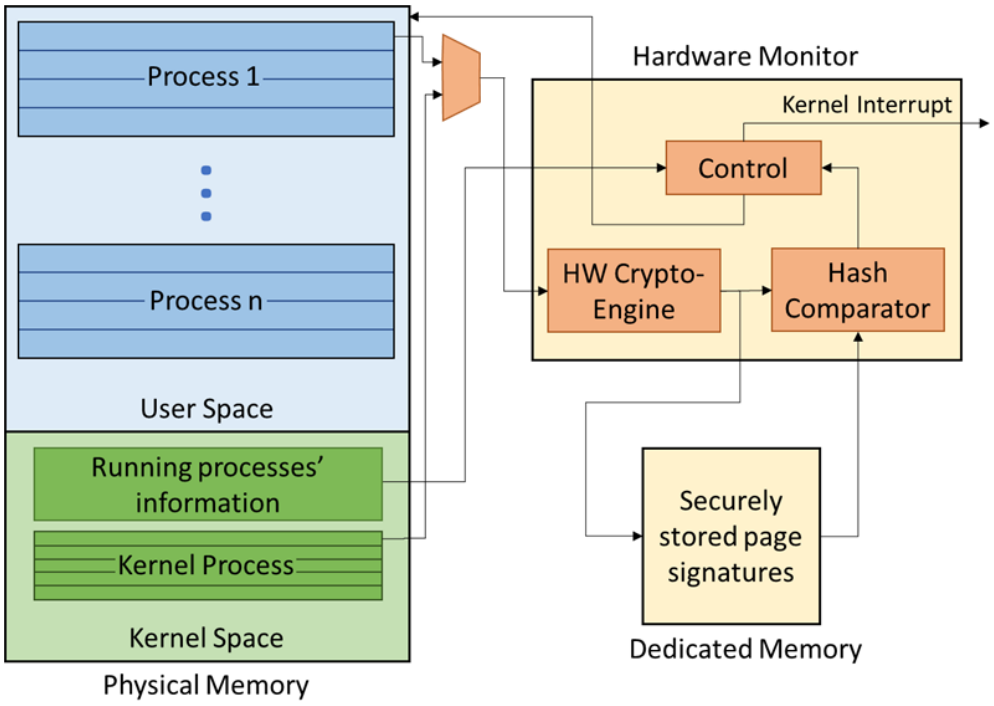


Figure 5.3: A modified architecture to support kernel process protection.

in memory to read and scan kernel-level processes' pages. Thus, our architecture is now able to check the integrity of specific kernel modules by performing the same technique of hash regeneration and checking at run-time. Of specific interest is our introduced security kernel-level driver which is responsible for interfacing between the kernel and the hardware monitor. In addition, it is imperative to protect kernel-level processes that perform memory management and page mapping/allocation from potential malicious code modification attacks as these kernel-level processes and drivers all play an integral role in our presented code integrity security model.

5.3 Implementation and Target Architecture

To implement our approach in an embedded system, we target a generic architecture similar to the one shown in Figure 5.4. Our presented approach is implemented in custom hardware and interfaces directly with the processor and its main memory. The operating system running on the processor is modified to include a kernel-level driver that interfaces with our custom hardware. In turn, the custom hardware includes components that control the interface with the processor and allow for direct memory access to the main memory and cache. In addition, the hardware is responsible for generating run-time hashes of the process pages, comparing the hashes to the golden references and interrupting the kernel in the case of a hash mismatch.

It is noteworthy to mention that the custom hardware could be an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). Obviously, each

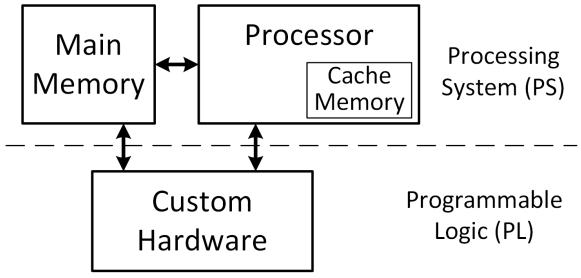


Figure 5.4: A generic embedded systems architecture of our presented approach.

of the different architectures provide distinct advantages and disadvantages. For example, implementing our presented technique on an ASIC chip would provide faster hardware components at the expense of the modularity provided by FPGAs. In particular, having our architecture implemented on an FPGA provides the ability to perform different architectural decisions such as choosing different hashing algorithms during a product's lifetime depending on the needed security of the application.

Implementing our approach on a target architecture such as the one shown in Figure 5.4 with a target ARM processor running Linux presents some challenges. We highlight these challenges along with some of the assumptions we take in our target architecture in the following subsections.

5.3.1 Assumptions and Challenges

Our architecture aims to maintain system integrity after the system boots. Specifically, our presented approach is complementary to other techniques that ensure a secure boot process is performed. Thus, we assume that the target processor and operating system boot into a trusted known state before applications start running. In addition, we initially assume that the kernel has varying degrees of protection from attacks, and then we relax these assumptions by expanding our architecture to perform run-time detection of malicious code modifications to kernel processes as well (see Section 5.2.3). For example, we first assume that the kernel has a high to medium level of assurance of some kernel processes that perform basic tasks such as scheduling, memory allocation, etc. Moreover, we assume that the hardware underneath the operating system is secure and has been provisioned by a system integrator. Finally, our presented architecture assumes that the kernel can be slightly modified to interact with our hardware module to ensure access to information regarding running processes such as memory maps, compiler, linker and loader information.

To accommodate the implementation of our architecture alongside the Linux operating system, our work is adapted to comply with challenges pertaining to the Linux memory

management and virtual memory implementation [82, 83]. For example, details of the exact locations of the executable code including shared library code are implemented. In addition, different types of linking code executables are taken into consideration. For instance, our implementation of the presented security architecture is modified to seamlessly accommodate both static and dynamic linking. Finally, our technique takes into consideration the implementation of our architecture on embedded systems with Address Space Layout Randomization (ASLR) including caches and their associated coherency protocols [84].

5.3.1.1 Linux Memory Management and Process Memory Address Space

After a successful compilation of a target application, the generated executable (ELF) file of the application is scanned (see Figure 5.5). The program and section header tables in the ELF file are analyzed to extract the application's executable pages which need to be loaded to memory prior to run-time. Specifically, the `p_type` entry in the program header table, shown in Table 5.1, is scanned for the loadable segments in the code (indicated

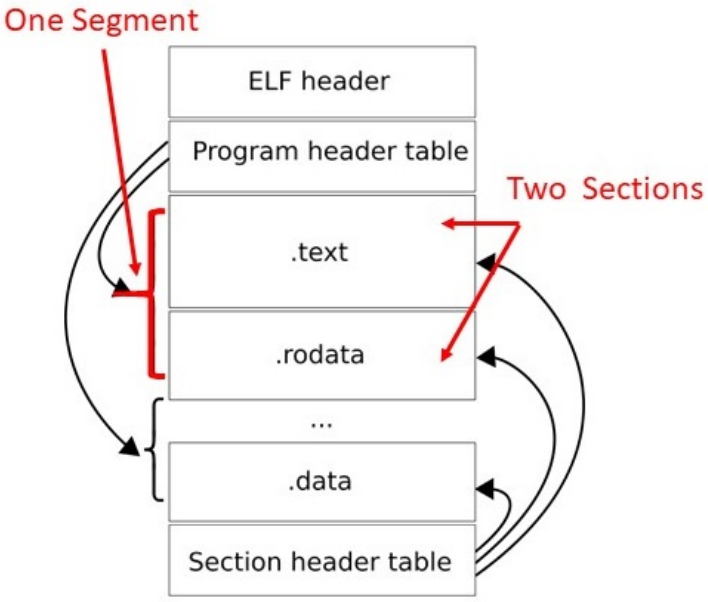


Figure 5.5: The ELF file structure showing the ELF header, program and section header tables and the different types of code and data segments and sections.

Table 5.1: The program header table showing the entries used to locate different segments in the ELF file.

ELF Address	p_type	p_offset	p_vaddr	p_filesz	p_memsz
...	...	...	...	...	...
0x94	PT_LOAD	0x00	0x8000	0xFC4	0xFC4
0xB4	PT_LOAD	0x1000	0x11000	0x138	0x13C
...	...	...	...	...	...

by `PT_LOAD`). For example, in this scenario the executable code segment found at offset `0x00` with a size of `0xFC4` in the ELF file is to be loaded into the virtual memory of the application at address `0x8000`. The `p_offset`, `p_vaddr`, `p_filesz` and `p_memsz` entries are used to map the loadable segments from the ELF file to the virtual memory when the application is to be run as shown in Figure 5.6. The extracted loadable pages are utilized to generate the golden hashes which are then stored in secure memory.

At run-time, to allow for seamless integration between the hardware and the Linux OS kernel, we use specific Linux function calls along with the process information pseudo-file system (`/proc`). For example, as shown in Figure 5.7, the `/proc/pid/maps` pseudo-

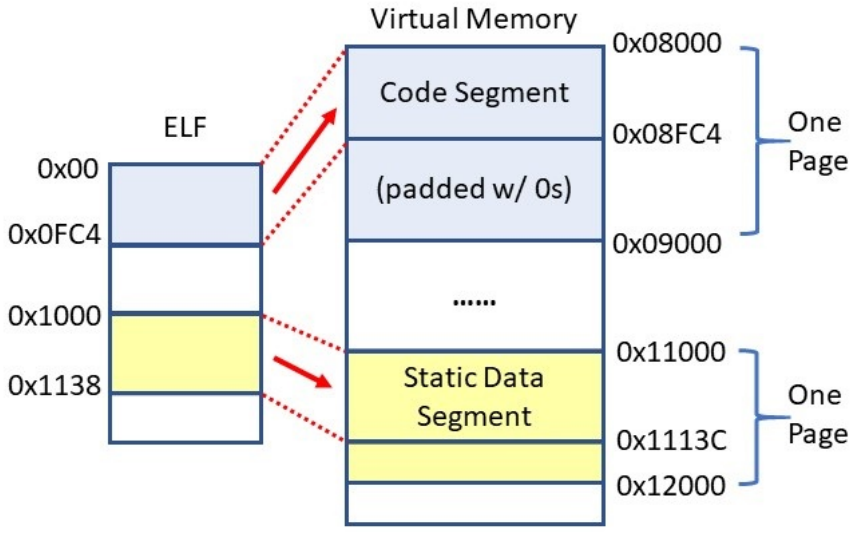


Figure 5.6: The ELF to virtual memory mapping showing the location at which code and data segments are loaded according to the page header table.

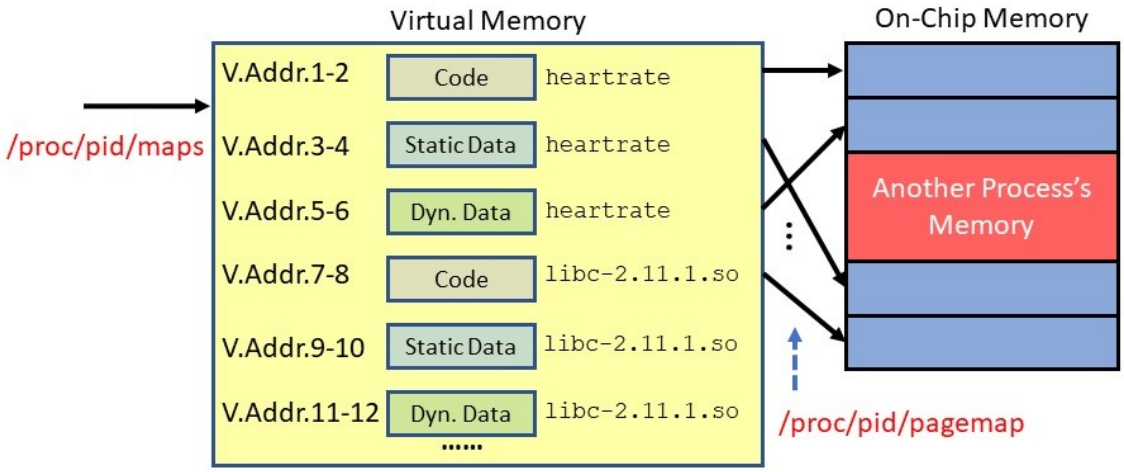


Figure 5.7: Extracting the virtual addresses of the application's pages and performing virtual to physical address translation using the Linux pseudo-file system.

file is used to extract the virtual addresses of executable pages and code segments of a running process along with the addresses of the shared libraries which the process calls. In addition, the `/proc/pid/pagemap` is used to fetch the physical address of a specific page's virtual address and extract the page's frame number. As mentioned in Section 5.2.2, the PFN is sent to the hardware monitor to perform on-the-fly hash calculations and comparisons of the monitored processes' pages. Therefore, our architecture supports systems that implement ASLR since virtual-to-physical address translations are performed at run-time.

5.3.1.2 Unmapped Page Regions

By analyzing the paging process of the Linux OS, we realized that executable code is typically placed in a set of contiguous pages. However, the size of the code in $bytes$ is rarely a multiple of the page size (4096 $bytes$ in our case). Thus, in most cases, the last page of an executable code would have some unmapped regions. To allow for correct hash generation during compile-time as well as run-time, we devised a technique where the unmapped page regions are masked with a set of binary zero values before being passed through the hash generator as shown in Figure 5.8. We start by dividing the page to a set

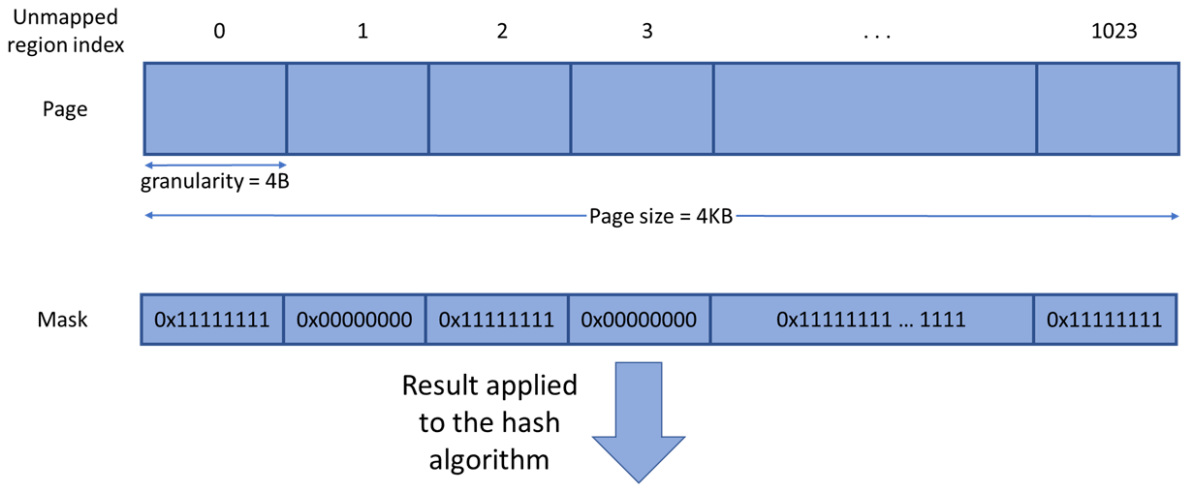


Figure 5.8: Our presented technique to handle unmapped page regions in an executable binary. The page is divided into equally sized regions where unmapped content is zeroed out before the page is sent to the hash algorithm.

of regions according to a predefined granularity. The granularity can be set per application to allow for increased security at the expense of performance. For example, as shown in Figure 5.8, the granularity is set to $4\ bytes$. Thus, the page is divided into 1024 regions. If, for example, regions 1 and 3 are unmapped, the page is masked such that those regions' binary values are ignored and substituted by zeros prior to being input to the hash generator. This way, even if some of the unmapped bits in the page change at run-time, a correctly regenerated hash will match the golden stored hash of that page.

5.3.1.3 Dynamically-Linked Libraries

To determine the library dependencies inside an application's code, we used specific Linux commands during the compilation phase of the process's binary. For example, the command `objdump` is used to extract the private headers of an ELF file which are then scanned for dynamic library dependencies [82]. The shared object (`.so`) files of the corresponding libraries to be linked are then scanned to extract the libraries' executable page contents. The executable pages are then hashed, one page at a time, and stored in the secure database along with the application's native page hashes in a fashion similar to the one presented

in Figure 5.9. To allow for scalability; furthermore, to reduce any possible performance impact, the database is first checked to see if the shared libraries' page hashes have been previously created by another monitored application before regenerating new hashes. It is important to note that in Linux the actual addresses involved in calling library functions are masked through the indirection provided by the Procedure Linkage Table (PLT) and the Global Offset Table (GOT) in the ELF file [82]. The dynamic loader would use the information provided in these tables to resolve the address of the dynamically linked libraries at run-time. Therefore, the actual code segment of the application remains consistent between compile, load and run-time.

It is worth mentioning that to protect against attacks that try to insert malicious code resulting in new pages that are unaccounted for, our hardware monitor can be configured to trigger an alarm in the presence of an extra number of executable pages as compared to the ones in the original application's code. For example, if the monitored application

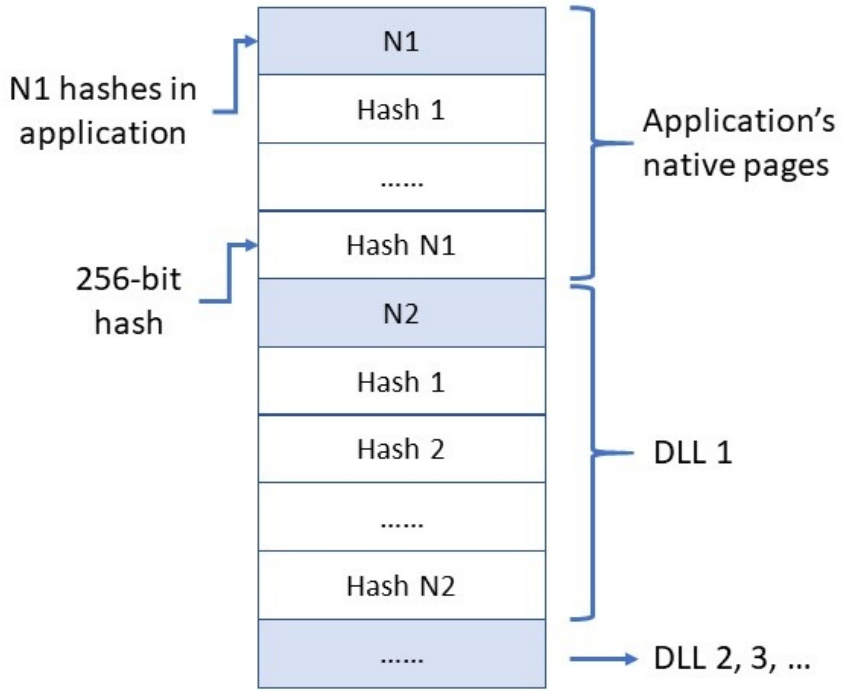


Figure 5.9: A sample file structure for storing the application's golden hashes along with the hashes of any dependent dynamically-linked libraries (DLLs).

has 10 golden hashes corresponding to 10 executable pages, the hardware monitor will be expecting to check for the integrity of only these 10 pages at run-time. Thus, if at any instance during the execution of the process, a new executable page is allocated, the hardware monitor will flag an alarm alerting the kernel that the new page does not have any corresponding golden hash. Therefore, our presented architecture currently limits the support of applications that allow for just-in-time (JIT) compilation and run-time code relocation [71].

5.3.2 Implementing Kernel-level Integrity Assessment

When implementing the architectural extension for assessing kernel-level process integrity, we focus on specific critical modules in the kernel. For example, in our current implementation, we monitor the kernel module responsible for the memory management of user-level tasks and applications (`task_mem`). In addition, we monitor the code segment of our kernel-level driver to ensure that the interface between the kernel and our hardware monitor is intact and protected from potential malware. Therefore, at kernel compile-time, we extract the code segments of the modules and drivers that need to be monitored. We then create the hashes of the executable pages of these modules.

Figure 5.10 shows the timeline that the user- and kernel-level process integrity architecture follows at run-time. Directly after the secure boot process ends and the kernel fully boots, the hardware monitor starts assessing the code integrity of the critical kernel modules and drivers. The hardware monitor uses a hardware-only accessible register to locate the addresses of the needed critical modules assuming that the kernel is loaded into the same location in memory at boot-time by the bootloader. For systems that have Kernel-ASLR (KASLR) [85] implemented, the hardware is informed of the kernel's loaded address after every system boot. Therefore, in these cases, our architecture is configured to use some Linux pseudo files such as `/proc/kallsyms` to locate addresses of the monitored critical kernel modules directly after the kernel boots and before any applications are run on the

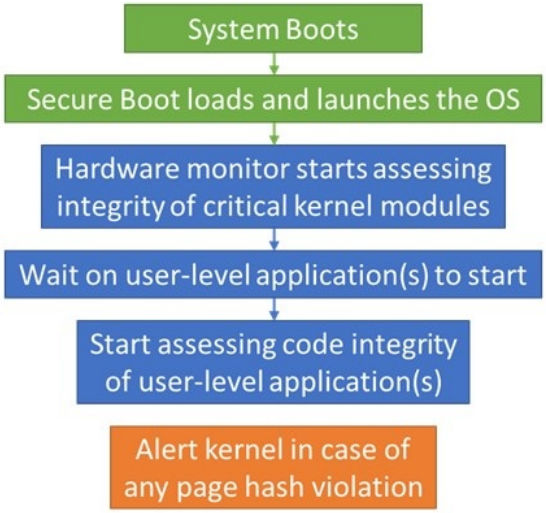


Figure 5.10: A timeline showing the steps performed at run-time by the user- and kernel-level process integrity architecture.

system. Next, the hardware monitor starts assessing the integrity of the pages of the critical kernel modules and drivers. Concurrently, the now protected kernel-level driver waits for the monitored user application(s) to start running. Once the operating system schedules one of the monitored processes, the hardware monitor starts assessing the integrity of both the recently scheduled user-level application and the kernel modules and drivers by regenerating the hashes of the monitored user- and kernel-level processes' pages and comparing them to the securely stored corresponding golden hashes. It is worth mentioning that if the kernel monitor triggers an alarm due to a hash violation of one of the kernel's pages, the hardware monitor takes action typically by safely shutting down the system under the assumption that the kernel has been attacked.

5.4 Summary

In this chapter, we present an overall approach, methodology and implementation of a novel hardware-based run-time code integrity checking architecture to detect malicious modification of application code. Specifically, we generate page-based hashes at run-time and compare them to securely stored golden hashes using a hardware monitor that is tightly

coupled to the processor's main and on-chip memory. We also present and implement a method that shows how to further expand our architecture to include assessing kernel-level process integrity. Our presented technique provides a way to assess the integrity of computation performed by a specific embedded device.

CHAPTER 6
EXPERIMENTAL SETUP, RESULTS AND ANALYSIS

6.1 Introduction

This chapter presents the experimental setup and simulations along with the hardware implementation and results of the two main architectures presented in this dissertation, namely, the hardware Trojan detection and the run-time code integrity architectures. In addition, performance, resource and power analysis of the architectures are reported.

The remaining sections of this chapter are divided as follows. Section 6.2 presents the experimental setup and reports and analyzes the simulation and synthesis results of the hardware Trojan detection architecture. Section 6.3 presents the experimental platform and setup along with the hardware implementation of the run-time code integrity architecture. In addition, experimental results are reported and analyzed demonstrating the effectiveness of the presented architecture on the target platform. Finally, conclusions are drawn in the summary in Section 6.4.

6.2 Hardware Trojan Detection

The following section reports and discusses the functional simulations and the synthesis results of our HT detection architecture which provides a suitable method for assessing sensor data integrity in embedded medical devices.

6.2.1 Experimental Setup

The digital components of our HT detection architecture presented in Chapter 4 and shown in Figure 4.10 were implemented using VHDL code, simulated using Mentor Graphics ModelSim version $10.6a$ and synthesized using Synopsys Design Compiler version J-

2014.09. We tested our architecture against multiple types of HT attacks on the ECG and/or BCG HF data that were captured over 60 seconds and sampled at a $2KHz$ rate from six different individuals. The subjects were healthy and at rest during the capture process. The human subjects measurements were approved by the Georgia Tech Institutional Review Board, and subjects provided written informed consent. When running the experiments, we emulated three different user health conditions that correspond to healthy individuals, individuals with minor heart problems and individuals with severe heart conditions.

The ECG and BCG sensors that we used in our experiments have analog output values that fall within a range of -0.9999 to 0.9999. In addition, the health monitoring application requires an accuracy of four significant digits after the decimal. In some cases in our architecture, values have to be squared and added, in which case the range of 0 to 1.9999 needs to be supported. Thus, to cover the range and provide the needed accuracy, we use a signed 16-bit fixed-point format with the most significant bit as the sign bit, the next bit as a representation of a value of 1 or 0, and the remaining 14 bits representing the fractional part of the number.

6.2.2 Hardware Trojan Design and Detection Analysis

We designed two different variations of each of the types of HT attacks presented in our threat model in Section 3.2.1. Specifically, we designed and implemented two HTs that target a single point in the architecture and another two HTs that target multiple points in the architecture. The attacks we implemented are numbered according to their type from 1 to 4. Figure 6.1 shows the locations at which these attacks target Chip 2 in our architecture. Attack types 1 and 2 are single attacks targeting a single point in the architecture. Attack types 3 and 4 are coordinated attacks simultaneously targeting multiple points in the architecture.

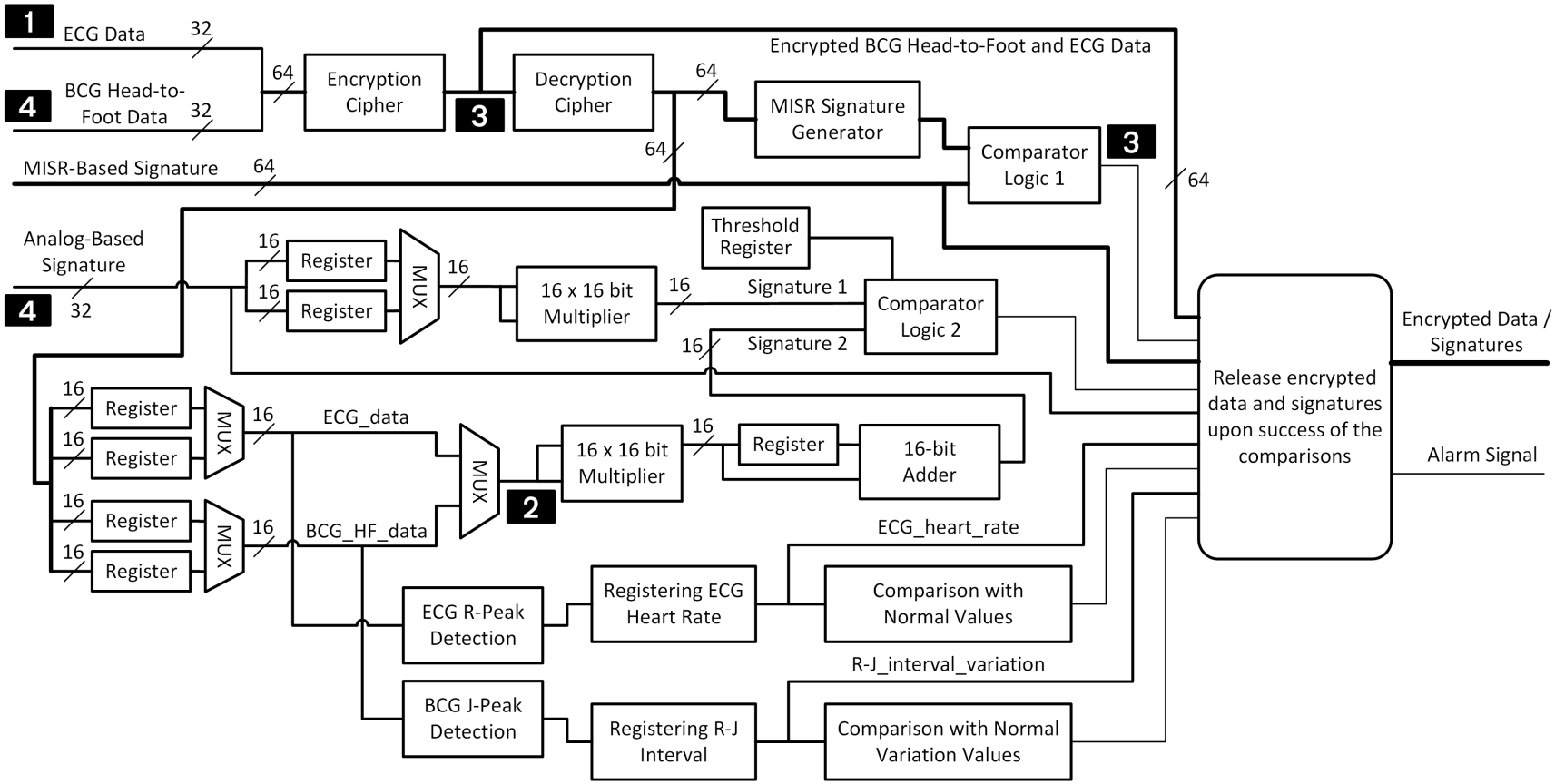


Figure 6.1: Possible HT attack types showing four different scenarios. Attack types 1 and 2 are single attacks targeting a single point in the architecture. Attack types 3 and 4 are coordinated attacks simultaneously targeting multiple points in the architecture.

6.2.2.1 Single Attack Type 1

Attack type number 1 is an example of an HT attack targeting a single point in the architecture, namely, the input data as soon as it appears on Chip 2 (Figure 6.1). Our work appears to be the first to provide an architecture able to detect such type of an attack. The difficulty in detecting this type of attack is that it happens before even any encryption or signature generation has been performed. Other types of signature-based HT detection techniques fail to detect this attack as their signature generation scheme relies on the input data [30, 33]. However, in our detection approach, since the initial signature is generated in Chip 1 (Figure 4.9), only our regenerated signatures in Chip 2 (Figure 4.10) will be affected. The comparison then with the golden signatures coming from Chip 1 will result in mismatches and the release logic will prevent the data and signature transmission out of the chip.

6.2.2.2 Single Attack Type 2

Attack type number 2 targets the intermediate data as it passes through the different internal modules in our architecture. Figure 6.1 shows an example of this type of attack where the HT tries to modify the output of the multiplexer, right before the data is fed to the squarer module for the analog-based signature (Signature 2 in Figure 6.1) regeneration. This results in the generation of an altered Signature 2, which when compared to Signature 1 (using Comparator Logic 2) results in a comparison mismatch and an alarm trigger. Hardware Trojans inserted to affect the output of the different modules have been studied earlier in the literature. Different detection techniques including signature-based ones were proven to be effective [30, 33, 34]. However, our combined architecture provides a unique way in detecting these attacks where multiple mechanisms operate synchronously to improve the assurance of the integrity of the user's medical data.

6.2.2.3 Coordinated Attack Type 3

HT attack type 3 attempts to initiate a coordinated attack simultaneously targeting two points in the architecture as shown in Figure 6.1. A detailed view of an HT of attack type 3 is introduced and presented in Section 4.7 and Figure 4.11. The HT trigger circuitry is connected to two payloads. In the example of attack type 3 shown in Figure 6.1, once the trigger is set, the HT simultaneously targets (i) the output of the encryption cipher by modifying it and (ii) the output of the comparator logic by forcing the result of the comparison to show a match even if the signatures at the input of the comparator logic do not match. Since this tiny HT remains always on once triggered, the comparator logic will always show a match regardless of input. The effect of both payloads result in the modification of the encrypted data and its passage undetected. However, as mentioned in Section 4.7, the comparator testing logic is inserted to detect such behavior and flag an alarm alerting a coordinated attack on the data and the comparator.

6.2.2.4 Coordinated Attack Type 4

Figure 6.2 shows a detailed view of an HT of attack type 4. The HT trigger circuitry in this case is also connected to two payloads. The first payload (Payload 1 in Figure 6.2) attacks the BCG HF input data, and the second payload (Payload 2 in Figure 6.2) attacks the analog-based signature (see Figure 6.1). A coordinated attack on the least significant bits of the BCG HF input data and the analog-based signature (as shown in Figure 6.2) will result in the modification of the data and, if the modified values result in $| \ Signature \ 1 - Signature \ 2| \leq threshold$, the HT operation might go undetected. However, the digital-based signature (MISR) and the physiological-based signature comparisons will detect the attack as shown in Table 4.1. In addition, it is unclear what ability the attacker would gain by changing the low order bits as these slight variations in the values of the inputs and the signature may also occur due to the analog nature of the application.

Another variant of attack type 4 is when the HT attempts to modify one of the high order

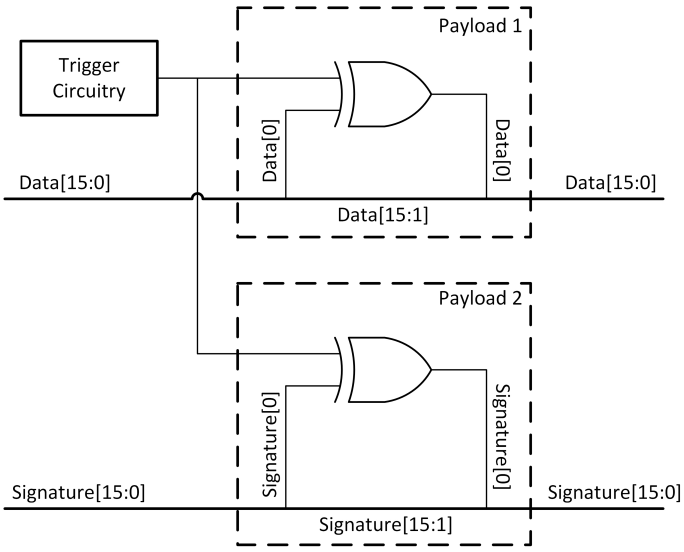


Figure 6.2: An HT attacking both an internal data bus in the design and the analog-based signature.

bits of both, the BCG HF input data and the analog-based signature. In this case, the attacker would have to exploit the vector sum relation between the ECG and BCG HF inputs to successfully modify both, the BCG HF data and the signature, in a way where the modifications pass undetected. However, this type of HT would require additional more complex circuitry (such as multipliers and adders) and would therefore fall beyond our threat model of a small sized HT as discussed in Section 3.2. Such types of attacks, and techniques for detecting HTs with large footprints (e.g., via power-based techniques in addition to others), have been well studied in the literature [24]. It is important to note that these prior HT detection techniques are complementary to our work and can be incorporated alongside our approach.

6.2.3 Simulation Results and Functional Verification

We simulated and tested multiple variations of the four types of HT attacks presented in Section 6.2.2 and shown in Figure 6.1. In our simulations, we set the *Threshold Register* of *Comparator Logic 2* in Figure 4.10 to a hexadecimal value of "0008" which represents a value of 2^{-11} in our fixed-point representation. The ECG peak threshold was set to

a hexadecimal value of "2000" which is equivalent to $0.5\ mV$. In addition, we set the ECG heart rate alarm ranges to represent "anomaly" when the values are below $30\ bpm$ (beats per minute) and above $150\ bpm$, "no anomaly" when the values are between $45\ bpm$ and $110\ bpm$, and "gray zone" otherwise [80]. The interval register was set to represent $400\ ms$. Normal R-J interval variation values were set to represent "anomaly" when the variation is above 50%, "no anomaly" when the variation is below 15%, and "gray zone" for variations inbetween [77]. Moreover, our HT trigger circuitry was configured to monitor for a specific occurrence of a BCG HF data sample, e.g., a hexadecimal value of "$12CF$" which represents approximately $0.29388\ mV$ in our fixed-point representation. Once that same value has appeared for 64 times, the HT was launched.

6.2.3.1 Simulation of Attack Type 1

To simulate HT attack type 1 (targeting a single point at the input of the state-of-the-art chip), we inserted HT logic similar to the one shown in Figure 3.1 targeting the input data as it arrives on chip. For example, in some of our simulations, we inserted HT logic that attacks the ECG input data as soon as it appears on the chip shown in Figure 6.1 by modifying the data such that one of the most significant bits in the 16-bit input was complemented. As mentioned in Section 3.2, the HT threat scenario that we consider in our work is triggered by some internal conditions or states. For our simulation purposes, the HT trigger waits on an attacker-defined number of occurrences of a specific input data. When the required condition is met, the trigger is set and the payload modifies the input data resulting in the modification of the functional behavior of the chip.

The modification of the input data resulted in modifying the encrypted and decrypted data which in turn lead to the modification of the MISR-based signature along with the regenerated analog-based signature (*Signature 2*) right before signature comparison. Once the maliciously modified signatures were compared to the original signatures coming from Chip 1 (Figure 6.1), *Comparator Logic 1* and *Comparator Logic 2* both declared mis-

matches. It is important to note that since the input data was altered by the HT, the values of the signatures at the input of *Comparator Logic 2* differ by an amount greater than the threshold (2^{-11}) and so *Comparator Logic 2*, similar to *Comparator Logic 1*, declared a mismatch at its output. In addition, the physiological feature extraction circuitry generated abnormal ECG heart rates and R-J interval variations confirming the mismatches of the signatures. Therefore, the release logic, monitoring the comparators' outputs, prevented the transmission of the altered encrypted data and asserted the alarm signal indicating the presence of the HT. All of this was verified through VHDL simulation using ModelSim.

6.2.3.2 Simulation of Attack Type 2

The simulation of attack type 2 was implemented in a similar fashion as attack type 1 due to the similarity in the attack. The major difference between this attack and that of type 1 is the place where the HT attacks. In attack type 2, the HT, once triggered, modifies the value at an output of a hardware block in the design. In our simulations, we performed multiple separate tests by inserting HT logic at the output of the different modules of the architecture.

For example, in one of our simulations, we inserted HT logic at the output of the multiplexer in the design as shown in Figure 6.1. This resulted in modifying a reasonably significant bit of the BCG HF data right before signature regeneration, leading eventually to an incorrect Signature 2. Once the regenerated signature (Signature 2) was compared to the digital version of the analog-based signature (Signature 1), *Comparator Logic 2* found that the signature difference exceeded the threshold and thus declared a mismatch so that the release logic prevented the transmission of the data and asserted the alarm signal.

Also, another simulation of the same attack type, this time at the output of the encryption cipher, confirmed the hypothesis presented in Section 4.3.2 for the need to decrypt the data and recreate the signature from the regenerated plaintext rather than directly from the input.

6.2.3.3 *Simulation of Attack Type* 3

When simulating attack type 3, the HT logic had to wait for the same trigger as in the previous attacks. When the trigger was set, the HT attacked two different points in the architecture as shown in Figure 4.11. Figure 6.1 shows the points at which we set the HT to attack in our simulations. *Payload 1* attacked the output of the encryption cipher and eventually led to modifications in the regenerated signatures. Simultaneously, *Payload 2* forced the output of *Comparator Logic 1* to show a match even when the compared signatures did not match.

It is important to note here that the comparator testing logic (described in Section 4.7) is periodically checking for this specific case. In our simulations, the periodicity was set to 16 iterations, i.e., *Test Mode* in Figure 4.12 is set to 1 after 16 sets of data have been processed through the architecture. *Test Mode* is asserted for only one clock cycle where the system is stalled and the comparator output is checked for legitimate operation.

Thus, the release logic might transmit altered encrypted data depending on when the HT is triggered. However, performing the testing periodically, can solve the problem if the sets of data between two consecutive tests (in our case, 16 sets) can be declared invalid if attack type 3 was detected (a multi-bit alarm signal can encode different types of alarm conditions, e.g., a specific bit encoding of the alarm could be used to indicate failure of the comparator testing logic).

In our simulation, we triggered the HT after six iterations of data have been processed. After an additional ten iterations and as soon as the *Test Mode* was asserted, the comparator testing logic read the result of the comparison and alerted the release logic to halt the transmission of the data while signaling the alarm.

6.2.3.4 *Simulation of Attack Type* 4

To simulate attack type 4, the HT was designed to wait for the same triggering mechanism and then attack the low order bits of both the BCG HF input data and the analog-based

signature shown in Figure 6.2. Our simulations show that such type of attack results in modifications to the values of *Signature 1* and *Signature 2*; however, these modifications are minimal (below the threshold of *Comparator Logic 2*) and are not detected by the analog-based signature testing mechanism. Fortunately, the MISR signature generation and testing method detects these types of attacks as any single bit flip in the original input to the signature generator generally results in multiple bit flips in the generated signature and therefore is detected by *Comparator Logic 1*. In addition and as expected, the physiological features (ECG heart rate and R-J interval variation) indicated minor alarm severity (gray zone) since the modifications in the features were minimal and not conclusive by themselves. However, when coupled with the mismatch indicated by the MISR comparator logic, the release logic was able to confirm the possibility of an HT attack and assert the chip's overall alarm signal.

6.2.3.5 Timing and Efficiency of Attack Detection

To compare the behavior of the different signature generation and testing techniques described in Sections 4.3 through 4.5, we reran the same types of HT attacks while varying the bit location which the HT inverts from the most significant bit (MSB) to the least significant bit (LSB). The aggregated results of the average time taken to detect an HT for each of the three techniques are presented in Figure 6.3. It is important to note that for the physiological features-based technique, an HT is considered to be detected if the alarm signal status indicates an anomaly or a gray zone condition. This does not apply to the remaining two techniques (MISR-based and analog-based) as these two techniques have only two alarm severity levels, a signature match and a signature mismatch status.

The experimental simulation results showed that the analog-based signature technique was the fastest in detecting HTs, i.e., within a few clock cycles, but the least accurate. The digital-based signature (MISR) technique was the most accurate and was able to detect all types of HTs. However, MISR-based signatures took a longer time to declare an alarm after

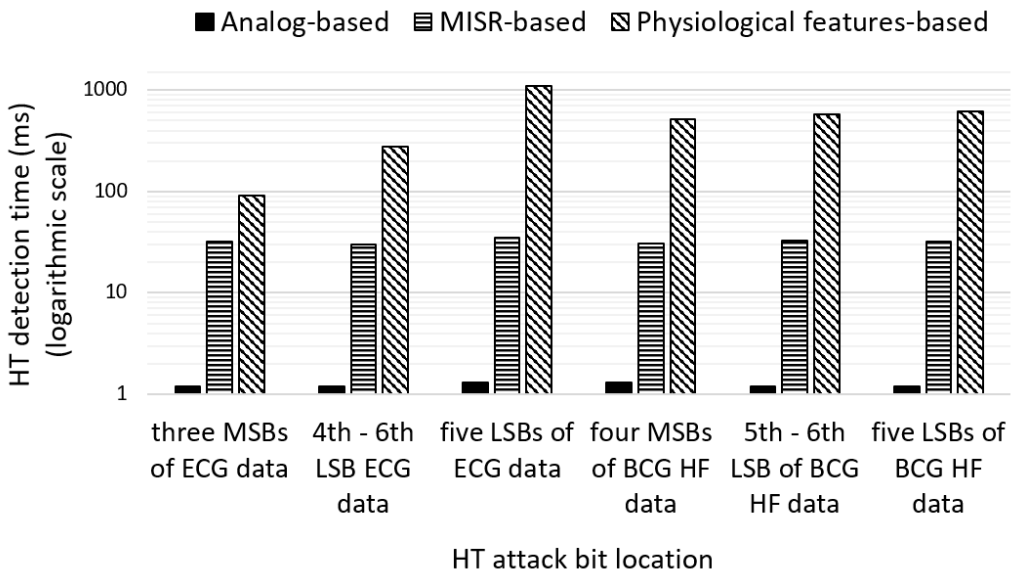


Figure 6.3: Time taken by each of the signature testing techniques to detect HTs targeting different data bit locations.

an HT was triggered. That is because they require compressing multiple sets of data and then checking for the integrity of the whole set. The accuracy of the physiological-based signature technique relied on the type of the HT attack. For example, for HTs that attack low order bits, a small portion were not reported (less than 5%) by the method. In addition, attacks on low order bits of the input data took a longer time to detect, while attacks on the most significant bits were detected at a much faster rate. Thus, the physiological-based signature technique proved to provide moderate HT detection accuracy, but was the slowest in HT detection primarily due to the fact that the technique inherently requires multiple sets of data to construct physiological features.

To better understand the effect of the different HT attacks on the physiological-based signature detection mechanism, we studied the alarm severity set by our architecture. We also studied the time taken for the first alarm to be set indicating the presence of an HT and the percentage of time the alarm signal indicated a definitive anomaly versus a gray zone indicating possibility of an attack. We triggered the HT after 10 seconds of real time. The results are reported in Table 6.1.

Table 6.1: HT attack effects on the physiological-based signature detection architecture.

HT attack on	**Time taken to detect HT**	**Severity of alarm**	
		Heart rate alarm signal	**R-J interval alarm signal**
any of the three MSBs of the ECG data	92 ms	Anomaly: 99.68% Gray: 0.32%	Anomaly: 70% Gray: 30%
any of the 4^{th} MSB to 6^{th} LSB of the ECG data	277 ms	Anomaly: 76.63% Gray: 23.37%	Anomaly: 41.95% Gray: 58.05%
any of the five LSBs of the ECG data	1098 ms	Anomaly: 0% Gray: 67%	Anomaly: 4.67% Gray: 95.33%
any of the four MSBs of the BCG HF data	515 ms	not applicable	Anomaly: 67.39% Gray: 32.61%
any of the 5^{th} MSB to 6^{th} LSB of the BCG HF data	577 ms	not applicable	Anomaly: 42.72% Gray: 57.28%
any of the five LSBs of the BCG HF data	612 ms	not applicable	Anomaly: 5.03% Gray: 94.97%

The results show that when an HT attacks any of the three most significant bits of the ECG input data, the HT is detected, on average, within 92 ms. This shows that the HT detection happens long before a new set of feature data is extracted as features are extracted at approximately every heartbeat which is equivalent to $0.5s - 2s$. The heart rate alarm signal is asserted for the remainder of the simulation indicating an anomaly 99.68% of the time and indicating a gray zone 0.32% of the time while the R-J interval alarm signal indicated an anomaly 70% of the time and a gray zone 30% of the time.

When an HT attacks any of the fourth MSB to the sixth LSB of the ECG data, the alarm is triggered, on average, after 277 ms. This delay is due to the time needed to wait until the next feature data is extracted (R-J interval data). Thus, as soon as the new feature's value is calculated, the alarm signal is triggered. The heart rate alarm signal indicated an anomaly 76.63% and a gray zone 23.37% of the remaining time of the simulation while the R-J interval alarm signal showed an anomaly for 41.95% and a gray zone for 58.05% of the time.

When an HT attacks any of the five LSBs of the ECG data, the alarm signal was asserted after approximately 1 second which is on average equivalent to the time taken for a heartbeat to occur. Thus, an HT attack was detected only after a new feature value was calculated. This is expected, as HT attacks on low order bits will not significantly affect the data to induce a peak in the ECG signal and create an early alarm. The heart rate alarm signal predictably did not show any anomaly for the remainder of the simulation time but indicated a gray zone status for about 67% of the time. The R-J interval alarm signal, however, indicated an anomaly for 4.67% of the time and a gray zone for 95.33% of the remaining simulation time.

HT attacks on the BCG HF data led to similar results with respect to the R-J interval alarm signal. As shown in Table 6.1, the R-J interval alarm signal was triggered for all cases around approximately $570\ ms$ and, as mentioned earlier, this refers to the average time taken to generate new values of the physiological features. Thus, as soon as a new value is generated, an alarm is triggered indicating the possibility of an HT attack with varying confidence for each of the cases as shown in Table 6.1. The ECG alarm signal, as expected, is never triggered when the BCG HF signal is attacked since the ECG heart rate feature does not use the BCG HF signal.

As expected, the simulation results showed that combining all three signature-based techniques into one architecture provided the ability to successfully detect all the different types of HT attacks defined in the threat model in the fastest time possible (within few clock cycles). In addition, the combination of the physiological-based signatures with both the analog and digital-based signatures not only allowed for the rapid detection of HT attacks and hardware errors in medical devices, but also helped in distinguishing them from health problems as reported earlier in Table 4.1. Thus, the presented HT detection architecture in this research addresses some of the shortcomings of the similar prior work described in Section 2.2.2 by checking for the data correctness while trying to detect HT attacks at run-time. In addition, our results show that the devised technique is HT-size independent and

can catch ultra-small HTs, since it looks for the effects generated by an HT momentarily after the HT is activated rather than searching for the hidden HT inside a digital microchip.

6.2.4 Synthesis Results

The digital modules of our combined architecture were synthesized using the Synopsys Design Compiler version J-2014.09 for Linux and were mapped to the $NCSU\ 45\ nm$ Base Kit Library [86]. Table 6.2 shows the area results of the main modules of our design post synthesis. It is obvious that a significant area of the architecture is covered by the encryption/decryption and processing modules. The security modules that are inserted to regenerate and test for the integrity of the data consume, as expected, a significantly lower area.

To better show the area overhead imposed by introducing our HT detection technique, we compute the overall area usage of the digital chip containing only the processing hardware and encryption/decryption units and compare it to the overall area of our modified architecture which includes the HT detection circuitry. The results show that the overhead introduced by the physiological-based signatures mechanism is the lowest (around 4%) as-

Table 6.2: Area results of the major digital components of our HT detection architecture

Module	**Area (square microns)**	**Area (kGE)**
Encryption Cipher (PRESENT [65])	5517	2.939
Decryption Cipher (PRESENT [65])	5431	2.893
MISR-based Signatures Overhead	6172	3.288
MISR-based Signatures Overhead (shared with digital systems test)	3575	1.904
Analog-based Signatures Overhead	1839	0.98
Physiological Feature Extraction	3485	1.856
Physiological-based Sig. Overhead	954	0.508
Release Logic	2414	1.286

suming that the physiological features are part of the processing that is done on chip. The analog-based signature generation and testing mechanism came in second with an overhead of around 7% while the MISR-based signature generation and testing technique had an overhead of around 14%. It is to be noted that our architecture allows for the use of any of the presented techniques by themselves or any combination depending on the needed security of the application at hand and the available resource and power limitations. Clearly, a combination of the three presented signature generation and testing methods achieves the highest confidence in detecting and distinguishing HT attacks and hardware errors from health problems at the expense of a higher area and energy consumption.

It is also important to note that in our experiments, Chip 2 contained only encryption and decryption blocks. In more realistic scenarios, such a chip could contain other processing and transmission modules which require larger area. Our conclusion is that the percentage overheads reported earlier can be considered pessimistic as increasing the overall chip area would eventually decrease the overhead of our HT detection approach.

Our current design achieves a maximum clock frequency of $300MHz$. An analysis of the timing results show that the multiplier that is used in the generation of the analog-based signature (*Signature 2* in Figure 4.10) falls along the critical path of our architecture. We currently implement the squaring operations in our design using Synopsys DesignWare's combinational carry save array multiplier. As reported by Synopsys [87], this type of implementation has a delay of $3.25\ ns$. If the application requires a higher clock speed a designer can choose to map the multiplier's logic to other implementations. For example, DesignWare has a Booth-recoded Wallace-tree multiplier which has a delay of $1.6\ ns$ (for a 16-bit multiplier). In addition, DesignWare provides other options of pipelined and sequential multipliers. Choosing between these types of implementations allows the designer to make area versus delay trade-offs.

6.3 Run-time Code Integrity

The following section reports and discusses the experimental and performance results along with the resource utilization of our run-time code integrity architecture which provides a suitable method for assessing the integrity of computation provided by an embedded systems application such as a heart rate monitor.

6.3.1 Experimental Platform and Setup

To test the effectiveness of our run-time code integrity architecture, our experiments were set up targeting the Digilent ZedBoard Zynq-7000 ARM/FPGA SoC development board shown in Figure 6.4 [88].

The ZedBoard hosts an ARM Cortex dual-core A9 processor including a Zynq 7000-series FPGA from Xilinx. The board provides an AXI bus to help interface the processing system (PS) with the programmable logic (PL). It includes, a $256\ MB$ Quad-SPI Flash memory which is enough to store the needed bitstream designs and OS boot images. The DRAM available on the ZedBoard is a $512\ MB$ DDR3 memory and is used to run the target application process to test our security architecture.

Thus, the hardware components of our run-time code integrity architecture were implemented using a combination of VHDL and Verilog code, simulated using Mentor Graphics ModelSim version 10.6a and synthesized using Xilinx Vivado Design Suite version 2017.4 targeting the Xilinx Zynq-7000 FPGA. In addition, as a sample embedded systems application, we developed a heart rate monitor and sample malware targeting an embedded version of Linux (PetaLinux) provided by Xilinx running on top of the dual-core ARM-based Cortex A9 processor on the Zynq-7000 SoC [89]. Our heart monitoring application used the same electrocardiogram (ECG) data captured from the six different individuals as reported in Section 6.2.1.

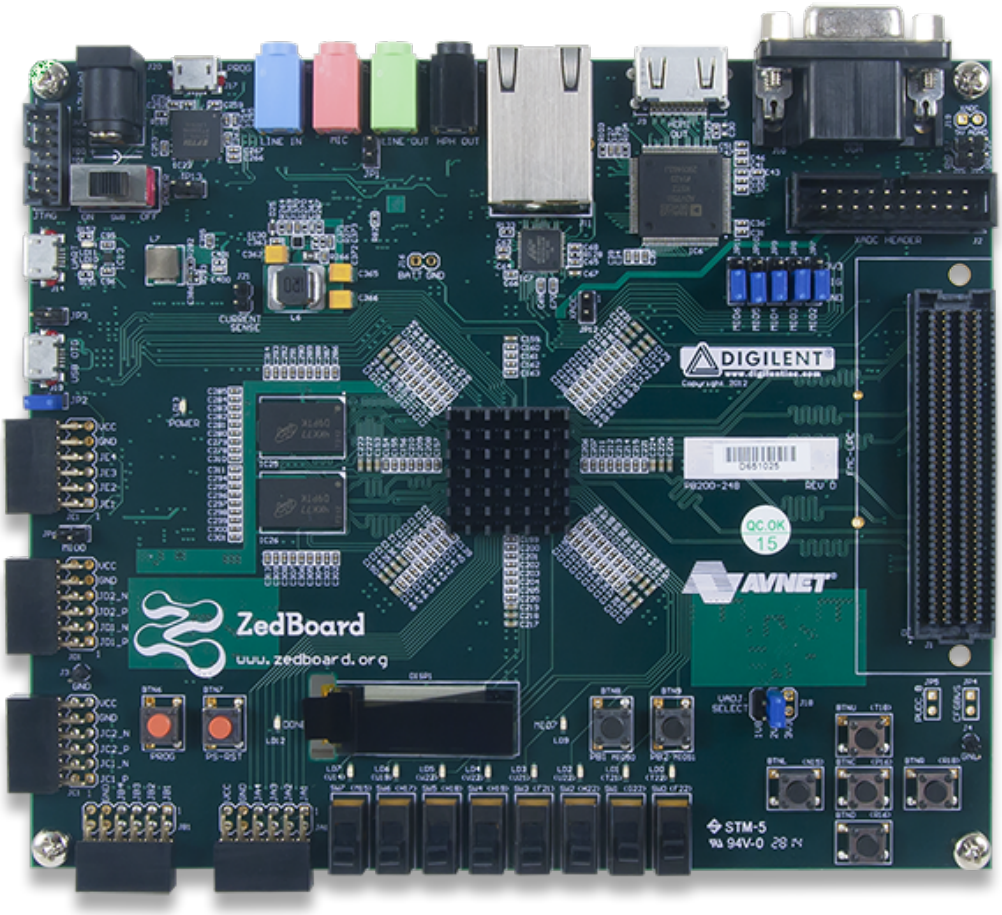


Figure 6.4: A top view of the Digilent Zedboard Zynq-7000 ARM/FPGA SoC development board [88].

6.3.2 Heart Rate Monitoring Application

Heart monitors typically measure a person's heart activity, such as rate and rhythm, and may take the form of a small embedded handheld or portable device. Figure 6.5 shows a similar example scenario to the one presented in Section 1.2 and Figure 1.2. However, in this case, the embedded medical system is attached to a treadmill in an exercise facility. The medical device monitors the heart rate activity of an individual while exercising. The person's electrocardiogram (ECG) signals are measured using grip-style dry electrode sensors [17] mounted on the handlebars of the treadmill. The captured ECG data is then used to find the person's heart rate. The calculated heart rate is displayed to the individual in real time. The embedded system is also internet connected to allow for syncing the user's

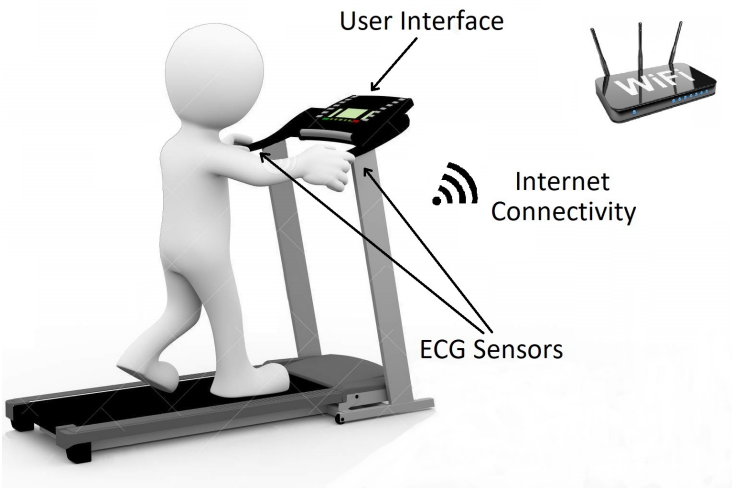


Figure 6.5: An example scenario of an embedded heart rate monitor attached to a treadmill machine in an exercise facility.

data with the cloud to perform long-term analysis and health diagnosis.

The heart rate monitor used in our experiments can be represented by the block diagram of Figure 6.6. An ECG sensor is used to capture the user's data which is then amplified and passed through a band-pass filter to improve the data's signal-to-noise ratio (SNR). The captured data is then fed to an analog-to-digital converter (ADC) which quantizes and samples the data at a 2 KHz rate. The filtered and sampled ECG data is then stored in

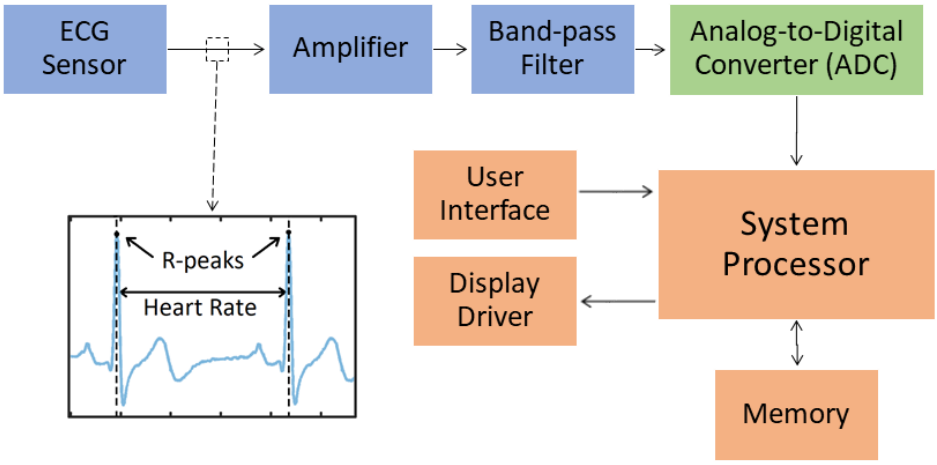


Figure 6.6: A heart monitor block diagram composed of electrocardiogram (ECG) sensors, amplifiers, filters, analog-to-digital converters (ADCs), a system processor, a memory, a processor interface, a display driver and a user interface.

the memory of the embedded device. The system processor runs an application to read the stored data for further processing and analysis. In our experiments, the sample heart rate monitoring application analyzes a user's ECG data to find the heart rate of the individual in real-time. Specifically, our software C code calculates the heart rates by scanning for consecutive ECG samples and finding the highest value (R-peak) within an ECG signal period. The time difference between two consecutive ECG R-peaks is registered and used to calculate a heart rate value. Figure 6.7 shows a sample result of the display of the user's heart rate monitor showing a current heart rate of $81.41\ bpm$.

6.3.3 Run-time Memory Corruption Malware

To test our run-time code integrity architecture against our target threat model described in Section 3.3, we developed in-house sample malware targeting the heart rate monitoring application described in Section 6.3.2. Our malware is assumed to have escalated privileges such that it is able to read and modify other user processes' memory contents. The assumption is that the attacker has enough resources to be able to capture the source and binary code of the target application – heart rate monitor in our case – to devise the attack. The developed malware sample is assumed to have avoided detection so far and attacks the heart rate monitor code at run-time and modifies the code contents of the application by

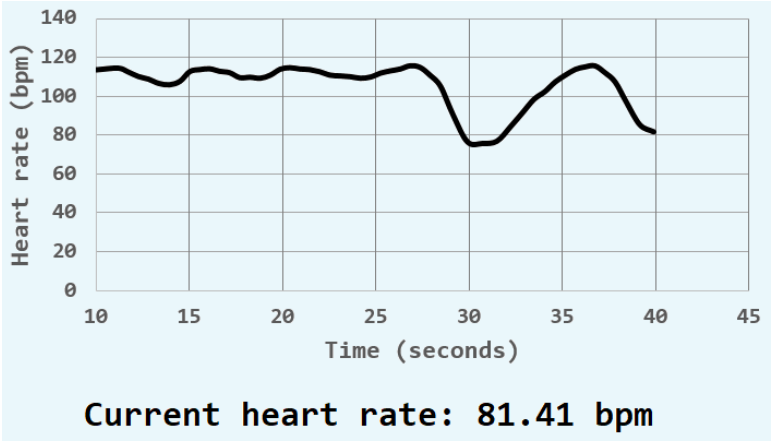


Figure 6.7: A heart rate monitor displaying a person's current heart rate with value of *81.41 bpm* along with the person's heart rate history over the past 30 seconds.

substituting a single instruction. We designed two variations of this malware. Specifically, in the first variation, a subtraction (`rsb`) instruction is replaced with a move (`mov`) instruction, and in the second variation the subtraction instruction is substituted by an addition (`add`) instruction.

Figure 6.8a shows a code snippet from the original heart rate monitoring application in the C language listing a series of instructions where the heart rate is calculated by subtracting two consecutive values of the ECG R-peak. The heart rate is then displayed to the user. The subtraction instruction is shown in boldface in Figure 6.8a. Figure 6.8b shows the equivalent binary code of the compiled and assembled instructions targeting the ARM assembly language. The corresponding assembly subtraction instruction (`rsb`) is shown in boldface in Figure 6.8b. Finally, Figure 6.8c shows the binary code of the attacked ap-

```
if (oldDataset > ECGThreshold) {              //R-peak threshold
     peakTime = currentTime-1;                //current peak time
     heartrate = (peakTime - oldPeakTime);    //calc heart rate
     printBeatDraw(heartBeatsNum++);
     printf("Heart rate: %.2f bpm\n", 60*2000/(double)heartrate);
     oldPeakTime = peakTime;      //set peak time as old peak time
}
```

(a)

```
.text:00008768  e51b2038        ldr r2, [fp, #-56]
.text:0000876c  e51b301c        ldr r3, [fp, #-28]
.text:00008770  e0633002        rsb r3, r3, r2
.text:00008774  e50b303c        str r3, [fp, #-60]
.text:00008778  e51b3020        ldr r3, [fp, #-32]
.text:0000877c  e2832001        add r2, r3, #1
```

(b)

```
.text:00008768  e51b2038        ldr r2, [fp, #-56]
.text:0000876c  e51b301c        ldr r3, [fp, #-28]
.text:00008770  e300378f        mov r3, #1935
.text:00008774  e50b303c        str r3, [fp, #-60]
.text:00008778  e51b3020        ldr r3, [fp, #-32]
.text:0000877c  e2832001        add r2, r3, #1
```

(c)

Figure 6.8: (a) Application code snippet in C. (b) Binary and assembly code snippet of heart monitoring application. (c) Binary and assembly code snippet of attacked application.

plication with the first variation of the malware where the `rsb` instruction is substituted by the `mov` instruction. The effect of this change masks the result of the calculation logic of the heart rate monitor leading to the generation of a perfectly normal heart rate value instead of the user's actual heart rate even if the user is having symptoms of a heart failure, thus potentially hiding the need for immediate medical help. The other variation where the `rsb` instruction is substituted by the `add` instruction would make the heart rate calculation display inaccurate values, potentially causing unnecessary efforts by the individual to seek medical help.

6.3.4 Hardware Implementation and Experimental Results

Figure 6.9 shows a diagram of our architecture implementation targeting the Zynq development board. To perform our tests, we generate at compile-time golden hashes of the executable pages (including any required dynamically linked libraries) of the critical kernel modules and drivers along with those of the monitored application (heart rate). The

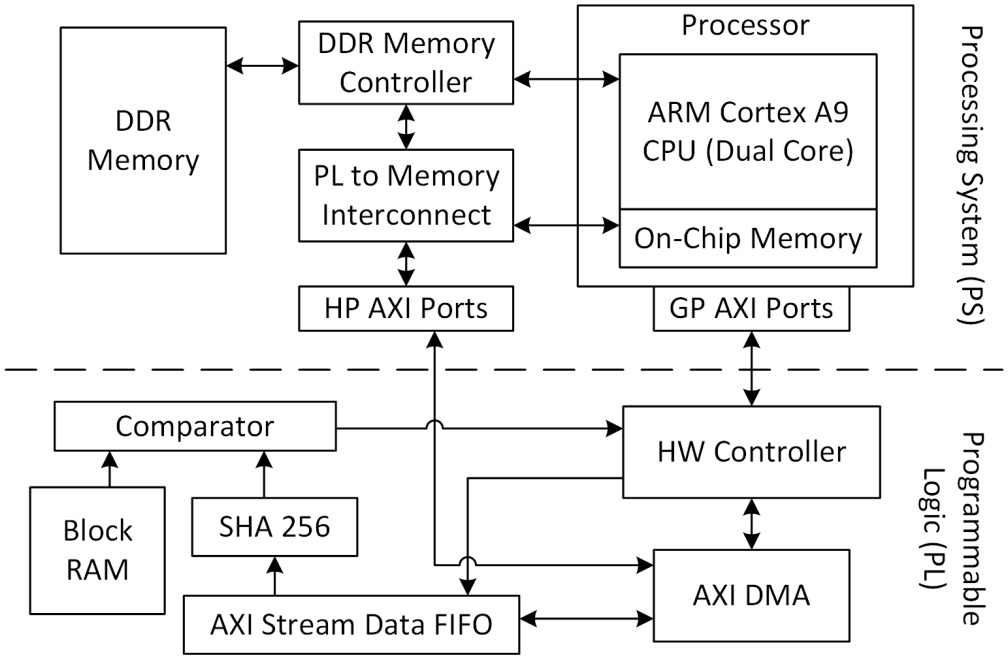


Figure 6.9: A detailed block diagram of the architectural implementation on the Digilent Zedboard.

golden hashes are stored in a secure memory (Block RAM in Figure 6.9) which is completely isolated from software (user and kernel space). The hardware monitor uses these hashes to check for the integrity of the user- and kernel-level processes' code at run-time.

To launch our run-time detection mechanism, as soon as the kernel boots, our hardware monitor accesses the pages of the kernel module `task_mem` and starts to periodically assess the pages' integrity. In the meantime, a kernel-level driver is started after the processor boots into a secure state. Once the kernel-level driver starts running, the hardware monitor is informed of the driver pages' addresses, and the integrity of the driver's code is now also continuously assessed. The kernel-level driver sets up the communication interface with the Programmable Logic (PL) in the FPGA and waits on the monitored process to start running. In our current implementation, we only monitor one application (the heart rate monitor); however, monitoring of multiple applications can be seamlessly integrated into our architecture. In fact, to ensure that this integration process can be easily done, we instantiated multiple runs of the heart rate monitor on different ECG data sets from two different individuals. Once an instance of the heart rate monitoring application is assigned a process id (*pid*), the kernel-level driver begins to continuously monitor the memory mapping of that process and extracts the physical addresses of the statically and dynamically linked executable pages. It then consecutively sends these physical addresses to the hardware monitor.

The hardware monitor in turn grabs the contents of the process pages from physical memory using the implemented AXI DMA streaming interface shown in Figure 6.9. Every process page is then fed through a hardware implementation of the SHA 256 algorithm [90], and the generated hash result is compared to the corresponding golden hash stored in the hardware isolated memory (Block RAM). Comparison results are then passed back to the kernel in the case of a failure as an interrupt. The kernel-level driver then takes control by halting the affected process. In the case where two instances of the heart rate monitor are simultaneously run, the kernel-level driver would consecutively send the physical addresses

of the pages of both processes to the hardware monitor for it to continuously assess the integrity of all the running monitored processes along with the memory management kernel module and the kernel-level driver.

6.3.4.1 Performance Analysis

To measure the effectiveness of our implemented architecture, we ran the heart rate monitoring application on a base design architecture excluding any of our presented security mechanisms and evaluated the performance of the system. The hardware was run at a clock frequency of $70\ MHz$ (the maximum achievable frequency with the current FPGA implementation of SHA256 [90]). The metric involved in our performance evaluation was the time taken by an instance of the heart rate monitoring application to read $120,000$ samples of ECG data (i.e., 60 seconds of ECG data since each second provides 2000 ECG data samples), process them, and continuously calculate and display the heart rate of the individual. This was done on the data sets of all six individuals. The aggregated timing results are shown in Table 6.3. We similarly re-ran the same heart rate monitoring application having the two malware variations execute and change the calculation of the application as shown in Section 6.3.3. The malware was randomly triggered using a combination of randomly selected system time and a keyword in keyboard inputs resulting in its execution

Table 6.3: Performance evaluation results comparing the baseline architecture with two versions of the modified process integrity architecture on the Digilent Zedboard development board.

Architecture	**Time to run** (ms)		
	Heart rate application	**Attacked application**	
		Malware 1	**Malware 2**
Baseline	$60,362$	$61,855$	$62,021$
Modified for user-level process integrity / Overhead	$60,671$ / 0.5%	$halted\ early$	$halted\ early$
Modified for user- and kernel-level process integrity / Overhead	$60,928$ / 0.94%	$halted\ early$	$halted\ early$

at different time instances during the application execution. Once triggered, the malware runs for a specific amount of time and then reconfigures the memory back to its normal state trying to mask the damage done. Since in this test we are not running our designed security mechanism, the malware was able to fully execute. With the malware running, we again measured the time taken by the heart rate monitoring application to perform the same tasks. The results are reported in Table 6.3.

Finally, we re-ran the application in the two different cases described earlier; however, this time the architecture was built and modified to introduce our hardware-based monitoring approach for assessing process integrity by modifying the PetaLinux kernel to insert our drivers and configuring the FPGA bitstream to implement our hardware monitor as shown in Figure 6.9. The performance of two versions of the modified architecture in terms of the aggregated time taken to execute the heart rate monitoring application on data from six different individuals is shown in Table 6.3. The first modified architecture included monitoring user-level processes only, while the second modified architecture included monitoring both user- and kernel-level processes and drivers. The results show that both versions of our architecture introduce minimal overhead on the performance of the ARM processor, specifically since the actual monitoring is only happening in hardware, and the software (Linux kernel driver) is minimally involved. In fact, the kernel-level driver is only carrying out the process of performing virtual to physical address resolution and sending the addresses to the hardware monitor. Therefore, continuously monitoring the page hashes of the application does not impact the target processor's performance.

Another critical metric that defines the effectiveness of our architecture is the time it takes our hardware monitor to detect the change in memory contents and report that change to the kernel. This was measured on our FPGA platform by starting a timer once the malware was triggered and calculating the time taken by the hardware to trigger an interrupt into the kernel. Figure 6.10 shows a sample of the output showing the monitoring of the heart rate application, the launch of the malware and the time taken by the user-

```
Starting rcS...
++ Mounting filesystem
FAT-fs (mmcblk0p1): Volume was not properly mounted. Some data may be corrupt. Please run fsck.
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting ssh daemon
random: sshd: uninitialized urandom read (32 bytes read)
rcS Complete
zynq> cd /mnt/
zynq> ./start_monitor.sh &
zynq> Monitor:    Monitoring started
Monitor:    Waiting for heartrate to start
zynq>
zynq>
zynq>
zynq> ./heartrate ECG_data.txt > heartrate_results.txt &
zynq> Monitor:    Started monitoring heartrate with process ID:  800

zynq>
zynq> ./my_sample_malware 1 800
Malware started at: 90266707 microseconds
Monitor:    Malware detected at: 90267017 microseconds
Monitor:    Killing process with PID: 800
[2]+  Killed                    ./heartrate ECG_data.txt 1>heartrate_results.txt
zynq>
zynq>
```

Figure 6.10: A snapshot of the output terminal showing the detection of the malware under the user-level process integrity architecture.

level process integrity architecture to detect the malware and kill the heart rate application. Similarly, Figure 6.11 shows an instance of the time taken to detect the malware attacking the heart rate application when the user- and kernel-level process integrity architecture is implemented. Notice that in Figure 6.11, the kernel monitor started as soon as the secure boot process finished to ensure the integrity of the monitored kernel-level modules directly after boot time. The security monitor then waited for the heart rate application to start. Once the application started, the monitor also started vetting the heart rate application's pages along with pages of the kernel-level module `task_mem`.

The aggregated results of the malware detection time for the two versions of our security architecture running on data from the six different individuals are reported in Table 6.4. This shows that our architecture is capable of detecting a malicious modification (as small as an instruction-level modification) to the heart rate application on average within $250 - 350\ \mu s$ if the architecture is monitoring user-level processes only, and within $700 - 800\ \mu s$ if the architecture is monitoring both user- and kernel-level processes. In other

```
Starting rcS...
++ Mounting filesystem
FAT-fs (mmcblk0p1): Volume was not properly mounted. Some data may be corrupt. Please run fsck.
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting ssh daemon
random: sshd: uninitialized urandom read (32 bytes read)
++ Running user script init.sh from SD Card
Boot process ended
Starting Kernel Monitor
rcS Complete
zynq> Started monitoring kernel module (task_mem)
Waiting for heartrate application to start

zynq> cd /mnt/journal/
zynq> ./heartrate ECG_data.txt > hrtrate_rslts.txt &
zynq> Monitoring of heartrate application with PID: 3151 has started

zynq>
zynq>
zynq> ./my_sample_malware 1 3151
Malware started at: 35783386 microseconds
Monitor:    Malware detected at: 35784132 microseconds
Monitor:    Killing process with PID: 3151
zynq>
[1]+  Killed                    ./heartrate ECG_data.txt 1>hrtrate_rslts.txt
zynq>
zynq>
```

Figure 6.11: A snapshot of the output terminal showing the detection of the malware under the user- and kernel-level process integrity architecture.

words, for a malware to be successful and circumvent our architecture (bypass the time-of-check to time-of-use TOCTOU race condition), the malware has to change the executable content of the application, perform its desired task and reconfigure the memory back to its normal state, all within this short time frame of less than a millisecond. Otherwise, our architecture will detect and flag the discrepancy resulting in a safe halt of the running application since the hardware monitor is continuously scanning all allocated physical pages

Table 6.4: Performance evaluation results showing the time taken to detect the malware after its triggered.

Architecture	**Malware variation**	**Time to detect malware (μs)**		
		Best	**Worst**	**Average**
User-level process integrity	1	220	543	287
	2	235	601	328
User- and kernel-level process integrity	1	635	985	720
	2	647	996	801

of the monitored application(s).

The periodicity of comparing the same page hash depends on the number of pages present in the application. In our experiments, the hash comparison for all the pages took around $600\ \mu s$ for the user-level process integrity architecture and $1\ ms$ for the user- and kernel-level process integrity architecture. These results were validated by the times taken to detect the malware attacks in the worst cases.

It is important to mention that the current detection times reported in Table 6.4 can be dramatically improved if the presented architecture is implemented on an ASIC chip with dedicated hardware resources as opposed to reconfigurable blocks on an FPGA. In addition, hardware parallelism can be introduced to allow for concurrent hash computation and simultaneous checking of multiple pages. Moreover, for ease of implementation, we currently implement parts of the DMA controller in software. Ideally, the controller will be fully implemented in hardware achieving faster malware detection. However, in our application scenario, heart rates are typically generated within $0.5 - 2.5\ sec$ and thus no further optimizations to the detection times are needed.

As opposed to the techniques presented in the literature, our architecture works on code that is present in memory and ready to be executed instead of relying on static instruction-based analysis and dynamic checking of expected code behavior. Thus, when comparing our performance results with similar work where code integrity is checked via hardware monitors [51–53], our approach presents a faster detection response with the ability to detect zero-day malware without imposing significant performance degradation on the embedded target processor. As shown in [51], using hashes of basic blocks for checking instruction integrity imposes a substantial overhead. For example, checking for the integrity of all the basic blocks of an application using the technique presented in [51] results in doubling the average clock cycles per instruction (CPI) of a processor. In contrast, our presented method eliminates this overhead by generating hashes for pages, thus significantly speeding up the integrity checking process. In addition, when comparing our architecture

to other software-related approaches, our work imposes fewer coding style limitations due to the reduced compile-time preprocessing that is required. Finally, our method provides a higher level of security as our hash generation and checking mechanism is completely isolated from software, and our technique only relies on basic kernel processes that are checked for their integrity using the same presented method.

6.3.4.2 Resource and Power Analysis

To study the impact of our code integrity architecture on resource and power utilization, we implemented both the baseline and the modified design targeting the same Digilent Zedboard Zynq-7000 ARM/FPGA SoC development board. Implementation results reported by the Vivado Design Suite showing area overhead imposed by our hardware architecture are presented in Table 6.5. In addition, estimates of the power utilization as reported by Vivado are shown in Table 6.6.

The reported results in Table 6.5 and 6.6 are for the architecture implementing both user- and kernel-level process monitoring. It is important to note that extending the architecture to support kernel-level process monitoring on top of user-level process monitoring did not significantly impact the area results. In fact, the difference between the two architecture versions can primarily be observed in the difference in Block RAM usage. As expected, the architecture implementing both user- and kernel-level process integrity requires more secure memory resources to store the golden hashes. Moreover, as the power utilization results in Table 6.6 show, the added security components do not incur signifi-

Table 6.5: Implementation results reported by the Vivado Design Suite showing the hardware overhead imposed by our architecture.

	Zynq-7000 FPGA Resource Utilization					
	LUT	**LUT RAM**	**Flip Flop**	**Block RAM**	**IO**	**BUFG**
Utilization	5091	1096	4391	8	8	1
Overall %	9.57	6.3	4.13	5.71	4	3.13

Table 6.6: Power estimates of the implemented design targeting the Zynq-7000 FPGA board as reported by the Vivado Design Suite.

Power (mW)								
Clocks	**Signals**	**Logic**	**Block RAM**	**IO**	**PS7**	**Dynamic**	**Device Static**	**Total**
12	15	13	1	6	1532	1579	146	1725

cant overhead since most of the energy is consumed by the Zynq processing system (PS7 in Table 6.6).

6.4 Summary

The experimental results and analysis presented in this chapter show that our designed and implemented HT detection and run-time code integrity architectures successfully detect different types of hardware and software malicious modifications attacking embedded medical devices.

Our hardware simulation and implementation results showed that the developed HT detection architecture was able to successfully detect the targeted types of HTs defined in the threat model including ones that target single points and multiple points in the architecture.

Our run-time code integrity architecture was able to successfully and quickly detect zero-day malware attacks on applications running on an embedded device such as a heart rate monitoring application. By focusing on physical memory integrity at the page-level for each process, the presented architecture reduces as much as possible any performance overhead by performing integrity checks in parallel using a dedicated hardware memory monitor.

Our synthesis results and hardware implementation show that it is feasible to implement both of our architectures while introducing minimal resource and performance overhead.

CHAPTER 7

EMBEDDED SYSTEMS SECURITY IMPACT AND OPEN RESEARCH QUESTIONS

7.1 Introduction

A majority of the hardware security techniques presented in this dissertation can be further improved and utilized to detect similar attacks on embedded systems in general. Therefore, the impact of this dissertation can touch a wide range of systems especially with the rapid growth of the Internet-of-Things (IoT) where resource constrained devices become highly interconnected making them a lucrative target for attackers. A major portion of these devices belong to the family of sensor nodes (Figure 7.1). Similar to medical and health systems, attacks on these critical sensor nodes and falsifying the captured data could also result in disastrous effects.

In this chapter, we present a modified version of the architecture described in Chapter 4 to help detect hardware-level attacks on a generic sensor node resulting in malicious modifications to the captured data and the on-chip computation [91]. As mentioned earlier in Section 4.2, our approach is similar in principal to the prover verifier technique presented in [72] where a trusted domain is used to verify the correct operation of an untrusted domain that acts as a prover of correct operation. However, our technique presents a dual-chip

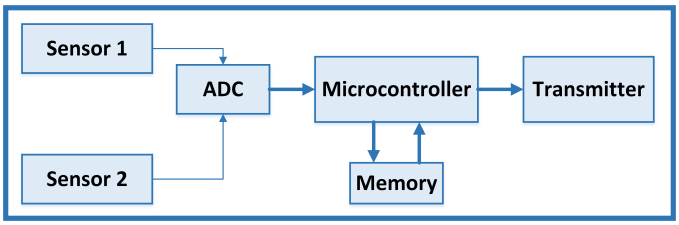


Figure 7.1: The main components of a generic wireless sensor node used to manage the capture and transmission of data.

approach to assess the integrity of the data as opposed to verifying the full specifications of the hardware components resulting in a simpler yet secure architecture that is able to guarantee with a high probability that the actual output of our design is as expected, in this case, proving that the data captured by the sensor node has not been maliciously modified on-chip. In addition, we present some open research questions and avenues for future work that could help improve the techniques presented in this dissertation to allow for better hardware and software security.

7.2 Chip-level Security Framework for Assessing Sensor Data Integrity

The scheme presented in this chapter builds on the technique discussed in Chapter 4 and on some of our published work in creating secure hardware by splitting the design into trusted and untrusted hardware domains [19–22]. However, in our previous attempts we relied on some assumptions where parts of the untrusted domain are considered to be trusted. In this section, we present a modified hardware framework for secure design with state-of-the-art untrusted chip fabrication. Figure 7.2 shows how a sensor node hardware could be split into a Trusted Verifier Chip and an Untrusted Prover Chip. Similar to our original architecture presented in Chapter 4, the trusted chip contains the sensor node components that are responsible for capturing analog data and digitizing the captured data. This microchip is manufactured in a secure and provisioned in-house fab. The untrusted chip contains the sensor node components that require cutting edge technology and is fabricated in a state-of-the-art facility. In addition, our modified framework allows for a way to provide secure reconfiguration of parts of the untrusted state-of-the-art chip so as to improve the overall security of the sensor node design.

7.2.1 Detailed Security Architecture

The design in the Trusted Verifier Chip is modified to add security components that help in verifying the integrity of the data as it passes through the untrusted part of the design.

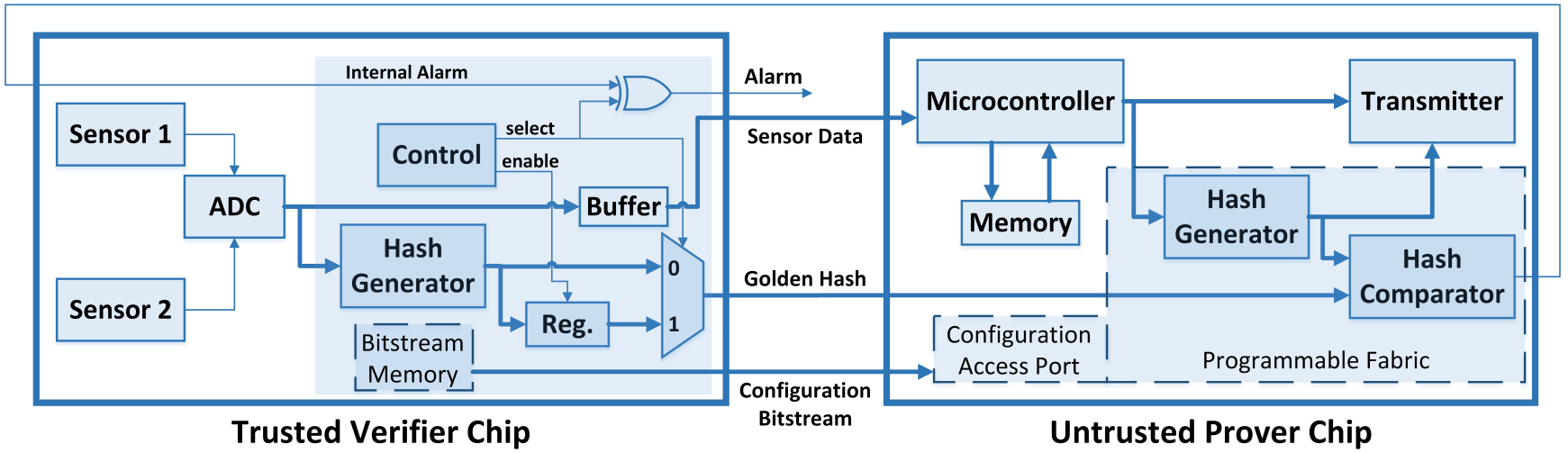


Figure 7.2: Our presented security model where the architecture is split into two domains. The first is demonstrated by the left chip which is manufactured in a secure and provisioned in-house fab and acts as a verifier of the second domain. The second is composed of an untrusted state-of-the-art chip that continuously acts as a prover at run-time.

Specifically, the Trusted Verifier Chip creates golden hashes of the data as it is being captured by the sensors. In addition, verification logic is used to test for the sanity of the components in the Untrusted Prover Chip. The added security components are highlighted by the shaded boxes in Figure 7.2.

The design in the Untrusted Prover Chip is modified to add the checking logic required to verify the integrity of the data and the respective computation while the data is being processed on chip. Specifically, a hash generator is used to regenerate the hashes of the captured sensor data before the data is transmitted out of the chip. The generated hashes are compared to the golden hashes at run-time and an *Internal Alarm* signal is sent from the Prover Chip to the Verifier Chip indicating the result of the comparison. The security components in the Prover Chip are checked for sanity of operation by the Verifier Chip at run-time.

During normal operation, the control unit in the Verifier Chip deasserts the *select* and *en* signals. The resulting datapath allows for the normal creation and passage of the captured data along with the golden hashes from the Verifier to the Prover. In addition, in this mode, the *Alarm* signal in the Trusted Verifier Chip will have the same value of *Internal Alarm* coming from the Untrusted Prover Chip.

Periodically, the control unit in the Verifier Chip sets the *en* signal to a logic value of '1' for one clock cycle to store a single hash of some sensor data then returns the system to normal operation. After another set amount of time and during the interval taken by the sensors and analog-to-digital converters to capture and digitize new data, the control unit sets the *select* signal to a logic value of '1'. At that instance, the datapath is reconfigured to pass the stored hash (a hash of some old data) along with new data to the prover chip. Ideally, the Prover Chip should return an asserted *Internal Alarm* indicating a mismatch between the regenerated hash and the golden hash passed from the Verifier Chip. This test phase verifies that the Hash Comparator block in the Prover Chip is behaving as expected. Specifically, since the Verifier sent some sensor data along with a hash that is known not to

match the data, the Verifier expects the Prover to send back an asserted *Internal Alarm* signal indicating an attack. However, if the *Internal Alarm* signal is not raised, the architecture will then flag the main *Alarm* signal.

7.2.2 Secure Hardware Reconfiguration

Our architecture can also be modified to include the ability to securely reprogram reconfigurable logic in the state-of-the-art Untrusted Prover Chip. As shown in the dotted boxes in Figure 7.2, a secure Bitstream Memory can be added to the design in the Trusted Verifier Chip so that various reconfiguration bitstreams can be securely stored and transmitted to the Untrusted Prover Chip for run-time reconfiguration of the hardware implemented in the chip. For example, the security modules (shown in the shaded box) in the Untrusted Prover Chip can be reconfigured at run-time to allow for more flexibility depending on the needed security and present hardware resources of the sensor node.

7.2.3 Attacks and Analysis

Similar to the discussed attacks in Chapter 4, the presented architecture in this chapter targets HTs that could be inserted to attack the primary inputs or any of the internal components of the untrusted chip. Attacks on the primary inputs, the microcontroller, the memory and/or the hash generator will be detected as any attempt to modify the data or the regenerated hash results in a failure when the comparison between the regenerated hash and the golden hash coming from the Untrusted Prover Chip happens. Attacks on the Hash Comparator block are detected by the mechanism provided in the Trusted Verifier Chip. Namely, if the attack attempts to fool our system by asserting a match even when the comparison is failing, the *Internal Alarm* sent back from the Prover Chip to the Verifier Chip will be tested for that specific failure case as described in Section 7.2.1. Finally, attacks on the transmitter resulting in modifications to the data or the hash can be detected by simply checking for the consistency of the sent data with the appended hash when the data is

received at the destination or during its transmission in the cloud.

Therefore, with minor modifications to the architecture presented in Chapter 4, we are able to provide a framework that helps in assessing data integrity in generic sensor nodes. Although we can guarantee with a high probability that incoming sensor data has not been altered while being received and processed by a state-of-the-art digital chip, we might not be able to fully guarantee that the outgoing sensor data has not been altered at all. However, our framework provides the means for the receiving end, e.g., the cloud, to perform specific checks to catch any such alterations.

7.3 Physical Layer Hardware Signatures as a Basis for Improved System Security

Our novel chip-level security framework presented in this dissertation could play a vital role as a basis for an improved security scheme that could be implemented for embedded devices with critical operations. The hardware technique along with its associated alarm signals can be used as run-time alerts for higher level policies and protocols. For example, our heart rate alarm signal presented in Chapter 4 and discussed in detail in Sections 4.5.1 and 4.5.3 could be divided into different severity levels and fed through to higher level software as one security metric that can be used to evaluate the trustworthiness of the underlying hardware components of a system.

In general, a hardware-level security architecture could then be devised out of multiple differently weighted security metrics and be fed to a higher level trust evaluator, such as the one presented in [92], to assess and determine the level of trust of a node in a network. In our heart rate example, the boundaries for the severity level of the alarm can be set as described in Section 4.5.4 and as shown in Figure 7.3. The set boundaries can be fine-tuned per individual to provide better accuracy.

The following subsection presents an example scenario showing how our hardware techniques can be used as a basis for evaluating the trustworthiness of the hardware components in a generic multi-layered trust environment to provide secure vehicle-to-vehicle

communication in autonomous cars.

7.3.1 Secure Vehicle-to-Vehicle (V2V) Communication

To provide better security for critical embedded systems, a multi-layer trust model can be implemented similar to the one shown in Figure 7.4. As the figure depicts, security techniques can be implemented at multiple layers such as (i) at the network, transport and application layers; (ii) at the physical and data link layers; and (iii) at the sensor and hardware layers. The techniques implemented at each of the layers will interact with each other at run-time to provide a better assessment of the security of an embedded system, in this case, an autonomous vehicle. For example, at the network layer, a network-based decentralized trust model [93] can be implemented where multiple nodes in a network can rate each others' trust. The network layer trust model would not only rely on the trust feedback provided from peers in the network, but also on metrics provided from the underlying layers.

The techniques presented in this dissertation can be implemented in the scheme presented in Figure 7.4. In fact, the alarm signals generated by our architecture can be fed to the upper physical and software layers to warn the system of possible malicious nodes in the network. In addition, our technique could benefit from feedback sent down from upper layers to improve the accuracy of detecting hardware attacks and errors.

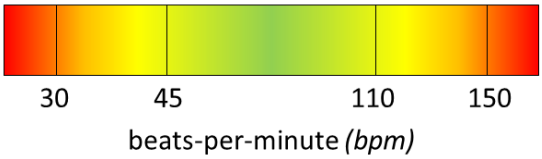


Figure 7.3: Different security alarm levels depending on the value of a patient's heart rate. A high level of trust is sent when the heart rate value ranges between 45 and 110 bpm (green zone). The trust level degrades as the heart rate value diverges (yellow to orange zone) from this normal range. A heart rate value below 30 bpm and above 150 bpm would show a low level of trust (red zone).

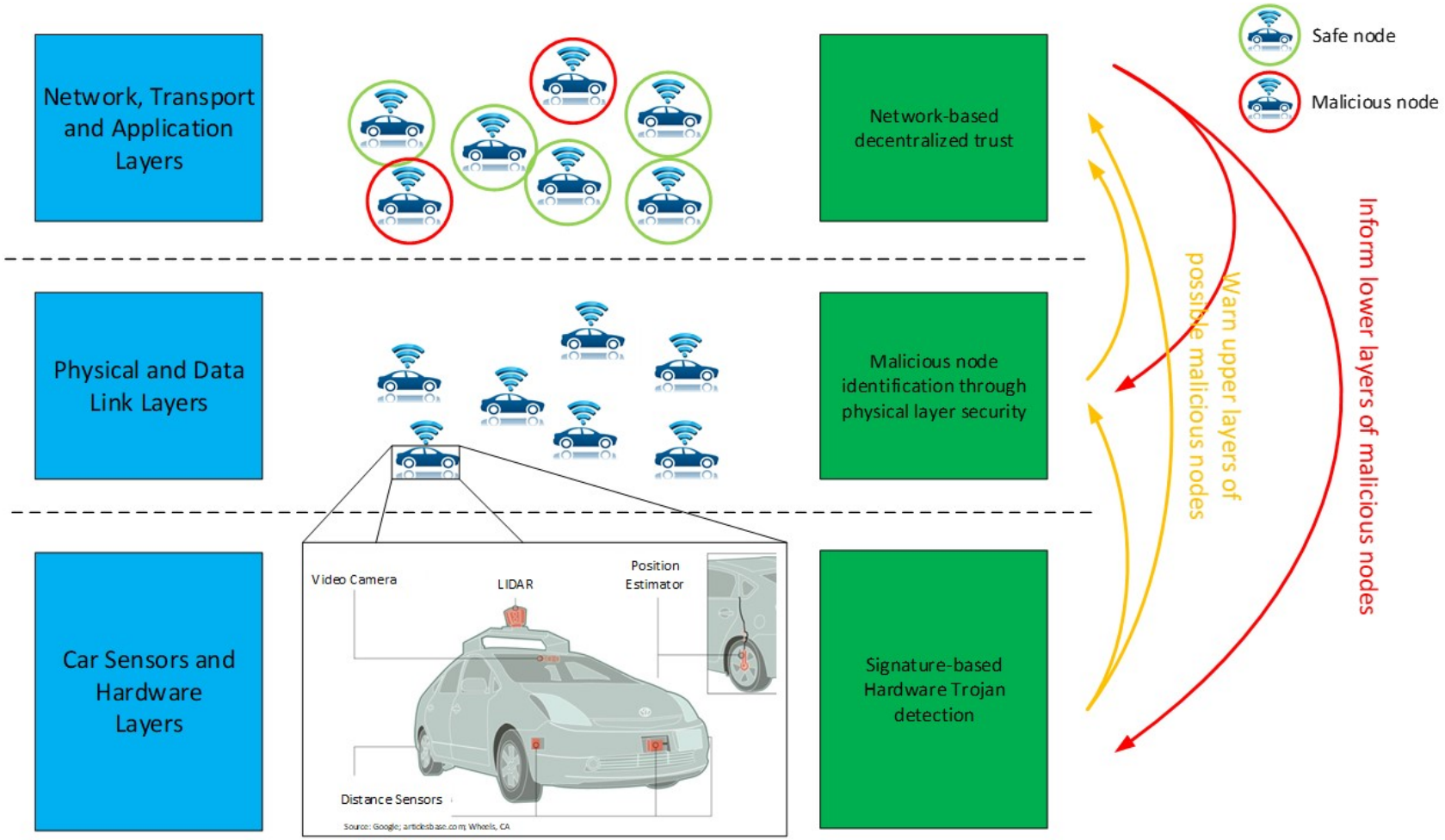


Figure 7.4: A multi-layer trust model showing interactions between the hardware, physical and software layers.

7.4 Open Research Questions

The research presented in this dissertation can be further improved to allow for better hardware and software security. In this section we present some open research questions and hint at potential solutions as avenues for future research.

7.4.1 Hardware Security

New techniques for attack localization and mitigation can be researched to analyze what can be done after a potential HT intrusion is discovered. In particular, a random error, e.g., due to thermal noise or x-rays flipping a logic bit value, could trigger a false alarm. A more complete approach would include a higher level protocol for identification of non-malicious random errors as a distinct class from errors which appear highly likely to be caused by an HT.

Another comment can be made about the transmission of the hardware signatures. Figure 4.10 shows that our architecture only encrypts the data and transmits the signatures out of the chip without explicitly encrypting them. Clearly, sending the signatures unencrypted might open an avenue of attack in later stages since an attacker may be able to exploit the unencrypted signatures to reveal information about the encrypted data. To prevent these types of threats, encrypting the signature can be done prior to transmission. Specifically, in a more complete view of a System-on-Chip (SoC) including logic for transmission packet formation, Figure 4.10 can be modified to include a similar logic to the one shown in Figure 7.5 to ensure only a properly encrypted bitstream is transmitted. Figure 7.5 shows multiple 16-bit analog-based signatures input to a FIFO buffer to form a block of 64-bit data

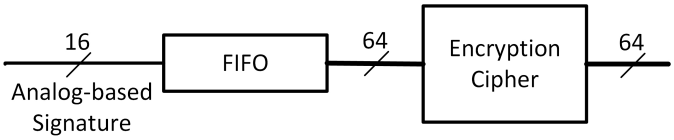


Figure 7.5: An example of encrypting analog-based signatures prior to transmission.

that then can be fed to an encryption cipher, such as PRESENT [65], to form an encrypted bitstream that is ready for transmission.

7.4.2 Software Security

The presented run-time code integrity architecture can be further improved to address some of the challenges introduced when supporting applications that allow for just-in-time (JIT) compilation and run-time code relocation [71]. One possibility is to include page hash generation both in software as well as in the hardware root-of-trust as an enrollment process for pages with code modifications. The challenge will be to keep this page hash generation process out of the hands of the adversary. Another challenge to be addressed is with cases of dynamic linking that involve object code modification at run-time by allowing for the insertion of new golden hashes at run-time.

Moreover, the security of our presented architecture can be improved by continuously searching for and addressing newer attack vectors that try to find and exploit any weakness that our architecture might have. For example, an adversary can try to craft an attack that takes advantage of the unmapped regions in executable pages to insert malicious code modules. One way to address such type of exploitation is by imposing a limitation on the mapping of executable pages where instead of masking the unmapped regions, the kernel would zero out these regions. This would improve the security of our architecture at the expense of a slight performance degradation due to a possible increase in paging overhead.

In addition, one interesting avenue to be further explored is looking into ways that not only focus on quick detection but also provide some corrective measures when possible. For example, a fast enough detection could help in preventing corrupted data from being used where the processor can be minimally stalled until the hash comparison is done.

7.5 Summary

In this chapter, we presented example scenarios for utilizing a generic chip-level security framework to detect malicious hardware modifications to state-of-the-art microchips that are widely used in embedded devices such as sensor nodes for capturing and transmitting data in the exploding world of IoT. Our architecture acts as a dual-chip approach composed of a prover and a verifier where the verifier continuously challenges the prover to maintain correct operation and detect any malicious modifications to the hardware. Preliminary analysis shows promise that our technique could be seamlessly integrated into a wider range of applications to help improve system security such as improving the level of trust in hardware in an example of a multi-layered trust model. The techniques presented in this dissertation can be further improved to help create an architecture that is more robust to both hardware and software attacks.

CHAPTER 8
CONCLUSIONS

This dissertation presents two different approaches for detecting run-time software and hardware malicious modifications in medical devices through the use of hardware signatures.

In the first approach, a hardware-based signature generation and testing architecture for detecting extremely small hardware Trojans in embedded medical devices is designed, simulated and synthesized. Three different techniques for signature generation are developed, namely, analog-, digital- and physiological-based signatures. In addition, an overall architecture combining all the three signature techniques is implemented. Simulation and synthesis results show that the developed architecture successfully detects the targeted types of HTs defined in the threat model with minimal area and performance overhead.

The second approach presents a novel hardware-assisted run-time code integrity checking architecture to detect malicious modification to application code running on an embedded processor in medical devices. The architecture relies on generating page-based signatures at compile time and storing these signatures in a root-of-trust. During execution time, a secure hardware monitor is used to continuously monitor the physical memory of the processor, grab a copy of the running application's pages and regenerate hardware signatures at run-time and compare them to the stored golden signatures. Our novel technique is implemented and evaluated on a Digilent Zedboard which holds an embedded ARM processor and a Xilinx Zynq-7000 FPGA. Results show that code integrity is achieved with minimal resource, power and performance overhead.

In summary, the software and hardware security architectures presented in this dissertation provide a basis for a methodology for protecting medical systems (e.g., heart monitors) from potentially malicious run-time code modifications and hardware attacks and errors.

Our architectures present novel ways of implementing dynamic code run-time integrity checking and hardware Trojan detection mechanisms using hardware-assisted signature generation and testing to assess the integrity of data and computation in embedded medical devices.

REFERENCES

[1] R. Johnson, *The Navy bought fake Chinese microchips that could have disarmed U.S. missiles*, Business Insider, 2011.

[2] J. Robertson and M. Riley, *The big hack: How China used a tiny chip to infiltrate U.S. companies*, Bloomberg Businessweek, 2018.

[3] J. West, T. Kohno, D. Lindsay, and J. Sechman, *WearFit: Security design analysis of a wearable fitness tracker*, IEEE Center for Secure Design, 2016.

[4] *Post-market management of cybersecurity in medical devices*, Center for Devices and Radiological Health, Food and Drug Administration, U.S. Department of Health and Human Services and Center for Biologics Evaluation and Research, 2016.

[5] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 461–491, Aug. 2004.

[6] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell, "Linux kernel integrity measurement using contextual inspection," in *Proc. of the ACM Workshop on Scalable Trusted Computing*, ser. STC '07, New York, NY, USA: ACM, 2007, pp. 21–29.

[7] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[8] K. Dunham, S. Hartman, M. Quintans, J. A. Morales, and T. Strazzere, *Android Malware and Analysis*, 1st. Boston, MA, USA: Auerbach Publications, 2014.

[9] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in *Proc. of the ACM Workshop on Scalable Trusted Computing*, ser. STC '09, ACM, 2009, pp. 49–54.

[10] T. Jaeger, R. Sailer, and U. Shankar, "Prima: Policy-reduced integrity measurement architecture," in *Proc. of the ACM Symp. on Access Control Models and Technologies*, ser. SACMAT, ACM, 2006, pp. 19–28.

[11] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Architectural support for runtime validation of program data properties," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 5, pp. 546–559, May 2007.

[12] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proc. of the 13th Conf. on USENIX Security Symposium - Volume 13*, ser. SSYM'04, Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16.

[13] G. Holmes, *Evolution of attacks on Cisco IOS devices*, Cisco Blog - Security, Cisco Systems, Oct. 2015.

[14] O. T. Inan, P. F. Migeotte, K. S. Park, M. Etemadi, K. Tavakolian, R. Casanella, J. Zanetti, J. Tank, I. Funtova, G. K. Prisk, and M. D. Rienzo, "Ballistocardiography and seismocardiography: A review of recent advances," *IEEE Journal of Biomedical and Health Informatics*, vol. 19, no. 4, pp. 1414–1427, 2015.

[15] M. Etemadi, O. T. Inan, L. Giovangrandi, and G. T. A. Kovacs, "Rapid assessment of cardiac contractility on a home bathroom scale," *IEEE Transactions on Information Technology in Biomedicine*, vol. 15, no. 6, pp. 864–869, 2011.

[16] G. K. Prisk, S. Verhaeghe, D. Padeken, H. Hamacher, and M. Paiva, "Three-dimensional ballistocardiography and respiratory motion in sustained microgravity," *Aviat Space Environ Med*, vol. 72, pp. 1067–1074, 2001.

[17] E. Richard and A. D. C. Chan, "Design of a gel-less two-electrode ECG monitor," in *Int'l Workshop on Medical Measurements and Applications*, 2010, pp. 92–96.

[18] D. D. He, E. S. Winokur, and C. G. Sodini, "An ear-worn continuous ballistocardiogram (BCG) sensor for cardiovascular monitoring," in *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2012, pp. 5030–5033.

[19] T. Wehbe, V. J. Mooney, D. C. Keezer, and N. B. Parham, "A novel approach to detect hardware Trojan attacks on primary data inputs," in *Proceedings of the 10th Workshop on Embedded Systems Security (WESS)*, 2015, 2:1–2:10.

[20] T. Wehbe, V. J. Mooney, A. Q. Javaid, and O. T. Inan, "A novel physiological features-assisted architecture for rapidly distinguishing health problems from hardware Trojan attacks and errors in medical devices," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017, pp. 106–109.

[21] T. Wehbe, V. J. Mooney, D. C. Keezer, O. T. Inan, and A. Q. Javaid, "Use of analog signatures for hardware Trojan detection," in *Proceedings of the 14th FPGAworld Conference*, 2017.

[22] T. Wehbe, V. J. Mooney, O. T. Inan, and D. C. Keezer, "Securing medical devices against hardware trojan attacks through analog-, digital-, and physiological-based

signatures," *Journal of Hardware and Systems Security*, vol. 2, no. 3, pp. 251–265, 2018.

[23] T. Wehbe, V. J. Mooney, and D. C. Keezer, "Hardware-based run-time code integrity in embedded devices," *MDPI Journal of Cryptography*, vol. 2, no. 3, 2018.

[24] M. Tehranipoor and F. Koushanfar, "A survey of hardware Trojan taxonomy and detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[25] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware Trojan attacks: Threat analysis and countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.

[26] S. Bhunia, M. Abramovici, D. Agrawal, P. Bradley, M. S. Hsiao, J. Plusquellic, and M. Tehranipoor, "Protection against hardware Trojan attacks: Towards a comprehensive solution," *IEEE Design Test*, vol. 30, no. 3, pp. 6–17, 2013.

[27] C. Lamech and J. Plusquellic, "Trojan detection based on delay variations measured using a high-precision, low-overhead embedded test structure," in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2012, pp. 75–82.

[28] S. Wei and M. Potkonjak, "The undetectable and unprovable hardware Trojan horse," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–2.

[29] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak, "Hardware Trojan horse benchmark via optimal creation and placement of malicious circuitry," in *DAC Design Automation Conference 2012*, 2012, pp. 90–95.

[30] T. F. Wu, K. Ganesan, Y. A. Hu, H. S. P. Wong, S. Wong, and S. Mitra, "TPAD: Hardware Trojan Prevention And Detection for trusted integrated circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 4, pp. 521–534, 2016.

[31] J. Francq and F. Frick, "Introduction to hardware Trojan detection methods," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, pp. 770–775.

[32] S. Moein, J. Subramnian, T. A. Gulliver, F. Gebali, and M. W. El-Kharashi, "Classification of hardware Trojan detection techniques," in *10th Int'l Conf. on Computer Engineering Systems (ICCES)*, 2015, pp. 357–362.

[33] A. Gbade-Alabi, D. Keezer, V. Mooney, A. Y. Poschmann, M. Stöttinger, and K. Divekar, "A signature based architecture for Trojan detection," in *Proceedings of*

the 9th Workshop on Embedded Systems Security (WESS), New Delhi, India, 2014, 3:1–3:10.

[34] D. Sullivan, J. Biggers, G. Zhu, S. Zhang, and Y. Jin, "FIGHT-metric: Functional identification of gate-level hardware trustworthiness," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–4.

[35] *Common Weakness Enumeration: A community-developed dictionary of software weakness types*, MITRE Corp. 2017.

[36] *Common Vulnerabilities and Exposures: The standard for information security vulnerability names*, MITRE Corp. 2017.

[37] J. Aycock, *Computer Viruses and Malware*. Springer, 2006.

[38] Y. Younan, W. Joosen, and F. Piessens, "Code injection in C and C++ : A survey of vulnerabilities and countermeasures," Katholieke Universiteit Leuven, Belgium, Tech. Rep., 2004.

[39] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, 1992.

[40] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[41] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, 4:1–4:40, 2009.

[42] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Vienna, Austria: ACM, 2016, pp. 356–367.

[43] C. Liu, M. Fan, Y. Feng, and G. Wang, "Dynamic integrity measurement model based on trusted computing," in *Int'l Conf. on Computational Intelligence and Security*, vol. 1, 2008, pp. 281–284.

[44] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu, "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)," in *20th Annual Computer Security Applications Conference*, 2004, pp. 82–90.

[45] S. H. Yong and S. Horwitz, "Protecting C programs from attacks via invalid pointer dereferences," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 307–316, 2003.

[46] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA)*, London, United Kingdom: ACM, 2007, pp. 196–206.

[47] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Int'l Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13, 2013.

[48] *ARM security technology - building a secure system using TrustZone technology*, ARM white paper, Apr. 2009.

[49] M. Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh, "Hardware-assisted detection of malicious software in embedded systems," *IEEE Embedded Sys. Letters*, vol. 4, no. 4, pp. 94–97, 2012.

[50] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 847–854, 2010.

[51] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 12, pp. 1295–1308, Dec. 2006.

[52] C. Li, D. Srinivasan, and T. Reindl, "Hardware-assisted malware detection for embedded systems in smart grid," in *IEEE Innovative Smart Grid Technologies - Asia (ISGT ASIA)*, Nov. 2015, pp. 1–6.

[53] A. K. Kanuparthi, R. Karri, G. Ormazabal, and S. K. Addepalli, "A high-performance, low-overhead microarchitecture for secure program execution," in *IEEE 30th International Conference on Computer Design (ICCD)*, 2012, pp. 102–107.

[54] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Piscataway, New Jersey: IEEE Press, 1990.

[55] *DRAFT FIPS PUB 202. SHA-3 standard: Permutation-based hash and extendable-output functions*, Information Technology Laboratory, National Institute of Standards and Technology, Online: `http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf`, 2014.

[56] *NISTIR 7896. Third-round reports of the SHA-3 cryptographic hash algorithm competition*, Information Technology Laboratory, National Institute of Standards and Technology, Online: `http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf`, 2012.

[57] *ATHENA database of ASIC results-ASIC rankings*, GMU Cryptographic Engineering Research Group, Online: `http://cryptography.gmu.edu/athenadb/asic_hash/rankings_view`.

[58] F. Gürkaynak, K. Gaj, B. Muheim, E. Homsirikamol, C. Keller, M. Rogawski, H. Kaeslin, and J. Kaps, "Lessons learned from designing a 65 nm ASIC for evaluating third round SHA-3 candidates," in *Proceedings of the 3rd SHA-3 Candidate Conference*, 2012.

[59] E. Kavun and T. Yalçın, "On the suitability of SHA-3 finalists for lightweight applications," in *Proceedings of the 3rd SHA-3 Candidate Conference*, 2012.

[60] E. J. Watson, "Primitive polynomials (Mod 2)," *Mathematics of Computation*, vol. 16, no. 79, pp. 368–369, 1962.

[61] L.-T. Wang, N. A. Touba, R. P. Brent, H. Xu, and H. Wang, "On designing transformed linear feedback shift registers with minimum hardware cost," Department of Electrical and Computer Engineering, The University of Texas at Austin, Tech. Rep., 2011.

[62] P. Koopman, *Maximal length LFSR feedback terms*, Department of Electrical and Computer Engineering, Carnegie Mellon University. `https://users.ece.cmu.edu/~koopman/lfsr/index.html`, Online; last accessed on 19 Sept. 2018.

[63] K. Kim, J. G. Tront, and D. S. Ha, "On hardware overhead in CMOS BIST designs," in *Proceedings of the IEEE Energy and Information Technologies in the Southeast*, 1989, 671–675 vol.2.

[64] R. Katti and R. Sule, "MISRs for fast authentication of long messages," in *Proceedings of the 16th Euromicro Conference on Digital System Design*, 2013, pp. 653–657.

[65] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, P. Paillier and I. Verbauwhede, Eds. Vienna, Austria: Springer, 2007, pp. 450–466.

[66] D. Fox, "Hardware security module (HSM)," *Datenschutz und Datensicherheit - DuD*, vol. 33, no. 9, pp. 564–564, 2009.

[67] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, 1st. Wiley Publishing, 2014.

[68] A. Matrosov and J. M. E. Rodionov D. Harley, *Stuxnet under the microscope*, Tech. Rep. ESET: Antivirus and Internet Security Solutions, Online; accessed on 30 March 2018, 2011.

[69] M. Prandini and M. Ramilli, "Return-oriented programming," *IEEE Security Privacy*, vol. 10, no. 6, pp. 84–87, 2012.

[70] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, Alexandria, Virginia, USA: ACM, 2008, pp. 27–38.

[71] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, "Language-independent sandboxing of just-in-time compilation and self-modifying code," *SIGPLAN Not.*, vol. 46, no. 6, pp. 355–366, Jun. 2011.

[72] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish, "Verifiable ASICs," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 759–778.

[73] F. Imeson, A. Emtenan, S. Garg, and M. V. Tripunitara, "Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation," in *Proc. of the 22nd USENIX Conf. on Security (SEC)*, USENIX Association, 2013, pp. 495–510.

[74] A. Q. Javaid, N. F. Fesmire, M. A. Weitnauer, and O. T. Inan, "Towards robust estimation of systolic time intervals using head-to-foot and dorso-ventral components of sternal acceleration signals," in *2015 IEEE 12th International Conference on Wearable and Implantable Body Sensor Networks (BSN)*, 2015, pp. 1–5.

[75] H. Ashouri and O. T. Inan, "Improving the accuracy of proximal timing detection from ballistocardiogram signals using a high bandwidth force plate," in *2016 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)*, 2016, pp. 553–556.

[76] R. F. Coughlin and R. S. Villanucci, *Introductory Operational Amplifiers and Linear ICs: Theory and Experimentation*. Harlow, United Kingdom: Pearson Education Limited, 1990.

[77] A. D. Wiens, M. Etemadi, S. Roy, L. Klein, and O. T. Inan, "Toward continuous, noninvasive assessment of ventricular function and hemodynamics: Wearable ballistocardiography," *IEEE Journal of Biomedical and Health Informatics*, vol. 19, no. 4, pp. 1435–1442, 2015.

[78] D. D. He, E. S. Winokur, and C. G. Sodini, "A continuous, wearable, and wireless heart monitor using head ballistocardiogram (BCG) and head electrocardiogram (ECG)," in *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2011, pp. 4729–4732.

[79] V. T. Jordanov and D. L. Hall, "Digital peak detector with noise threshold," in *IEEE Nuclear Science Symp. Conf. Record*, vol. 1, 2002, 140–142 vol.1.

[80] R. C. Schlant, R. Adolph, J. DiMarco, L. Dreifus, M. Dunn, C. Fisch, and *et al.*, "Guidelines for electrocardiography. A report of the American College of Cardiology/American Heart Association Task Force on assessment of diagnostic and therapeutic cardiovascular procedures," *Journal of the American College of Cardiology*, vol. 19, no. 3, pp. 473–481, 1992.

[81] M. D. Yu and S. Devadas, "Secure and robust error correction for physical unclonable functions," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 48–65, 2010.

[82] W. Mauerer, *Professional Linux Kernel Architecture*. Birmingham, UK, UK: Wrox Press Ltd., 2008.

[83] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Upper Saddle River: Prentice Hall, 2004.

[84] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04, New York, NY, USA: ACM, 2004, pp. 298–307.

[85] K. Cook, *Kernel address space layout randomization*, Linux Security Summit, `https://outflux.net/slides/2013/lss/kaslr.pdf`. Online; accessed on 19 May 2018, May 2013.

[86] *NCSU 45nm FreePDK™ process design kit*, Electronic Design Automation, North Carolina State University, `http://www.eda.ncsu.edu/wiki/FreePDK`.

[87] A. H. Syed, *Performance of different multipliers in the DesignWare building block IP*, DesignWare Technical Bulletin, Synopsys Inc.

[88] *Zedboard Zynq-7000 ARM/FPGA SoC development board*, `https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board`, Online; accessed on 12 July 2017.

[89] *Petalinux tools*, Xilinx, Inc. `https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html`, Online; accessed on 8 Aug. 2017.

[90] J. Doin, *SHA 256 hash core*, `https://opencores.org/project,sha256_hash_core`, Online; accessed on 7 Sept. 2017.

[91] T. Wehbe, V. J. Mooney, and D. C. Keezer, "A chip-level security framework for assessing sensor data integrity: Work-in-progress," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES '18, Turin, Italy: IEEE Press, 2018, 20:1–20:2.

[92] L. Liu, M. Loper, Y. Ozkaya, A. Yasar, and E. Yigitoglu, "Machine to machine trust in the IoT era," in *Proceedings of the 18th International Conference on Trust in Agent Societies - Volume 1578*, ser. TRUST'16, Singapore, Singapore: CEUR-WS.org, 2016, pp. 18–29.

[93] S. A. Soleymani, A. H. Abdullah, W. H. Hassan, M. H. Anisi, S. Goudarzi, M. A. Rezazadeh Baee, and S. Mandala, "Trust management in vehicular ad hoc network: A systematic review," *EURASIP Journal on Wireless Communications and Networking*, vol. 2015, no. 1, p. 146, 2015.

VITA

Taimour Wehbe was born in Al Bennay, Aley, Lebanon. He earned his Bachelor's of Science degree in Computer and Communications Engineering from the American University of Science and Technology in Achrafieh, Beirut, Lebanon in June 2011. In January 2012, Taimour moved to the United States to pursue his graduate studies. He earned his Master's degree in Electrical Engineering from Villanova University in Villanova, PA, USA in December 2013. In August 2014, he started pursuing his Ph.D. degree in Electrical and Computer Engineering (ECE) at the Georgia Institute of Technology in Atlanta, GA, USA.

During his Ph.D. studies at Georgia Tech, Taimour worked as a Graduate Teaching Assistant (GTA) and Graduate Research Assistant (GRA) in the School of ECE. As a GTA, he helped professors teaching undergraduate and graduate courses. He also served as an instructor teaching Circuits and Electronics for non-ECE students for several semesters. In 2016, he was selected as a recipient of the Outstanding ECE GTA award. As a GRA, Taimour worked in the Hardware/Software Codesign Group for Security advised by Prof. Vincent J. Mooney III. His research interests include hardware security and trust, embedded and medical systems security, reconfigurable and fault-tolerant computing, and Systems-on-Chip. He also helped supervise undergraduate research projects in hardware security.

In January 2017, Taimour was selected as one of two inaugural Cybersecurity Fellows at Georgia Tech by the Institute for Information Security and Privacy. In April 2018, he was selected as one of two students to attend and represent Georgia Tech at the RSA Conference 2018 as an RSA Conference Security Scholar. Taimour has published over 10 journal articles and refereed conference papers and has attended and presented his research findings in multiple conferences in the United States and around the world. He has been an active member in the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinary (ACM) for more than nine years.